

# REPORT

## [크리스마스 미로 게임]



•수강과목:	컴퓨터그래픽스 02분반
•담당교수:	서상현
•제 출 일:	2024.12.15
•학 부:	예술공학부
•팀 순서 :	19팀
	이윤재(20221453)
•구성원:	전혜진(20221464)
	)

## 목 차

<b>1.</b> 서론	2
<b>2.</b> 본론	3
<b>2.1.</b> 과제 개요 및 설계	3
<b>2.2.</b> 주요기능	3
<b>2.3.</b> 추가기능	3
<b>2.4.</b> 시행착오 및 해결과정	3
<b>3.</b> 결론	4

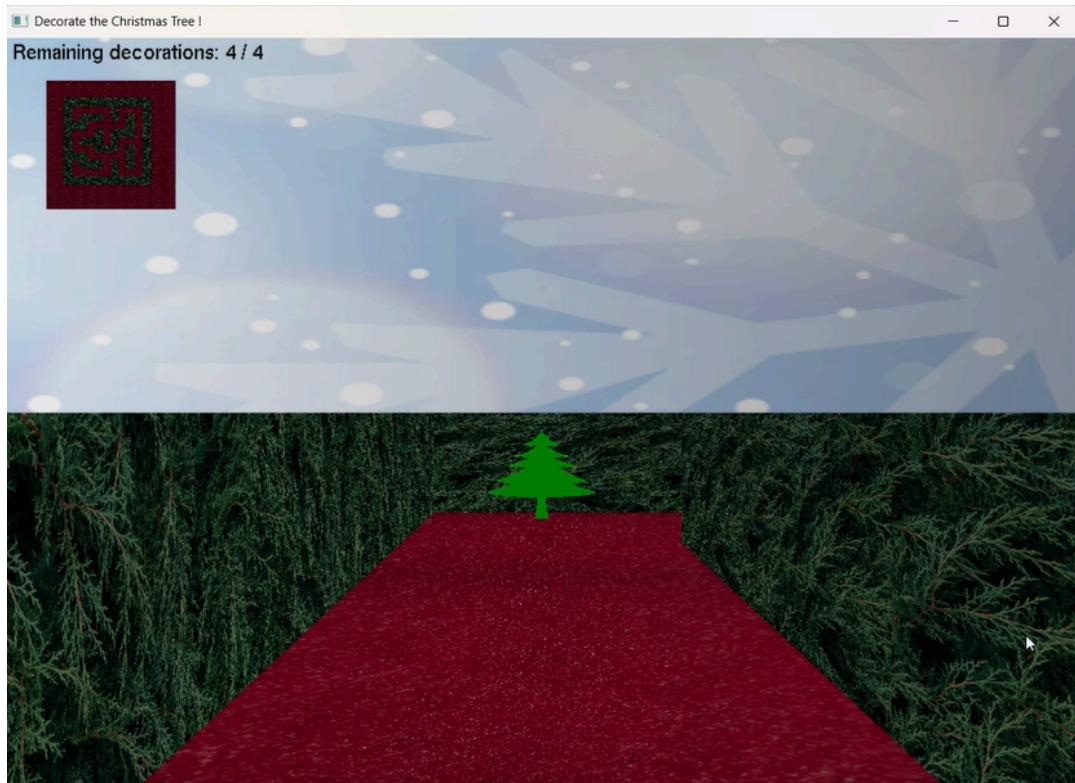
## 1. 서론

본 과제는 OpenGL을 활용하여 SOR 화면 출력에서부터 모델링 데이터를 저장하고 이를 미로에 불러와 게임 플레이 기능을 구현하는 과정을 다룬다. SOR 화면 구현, 미로 구성 및 오브젝트 배치, 카메라 설정 등 수업 시간에 배운 OpenGL의 기본 기능을 바탕으로 진행되었으며, 텍스처 매핑, 카메라 설정, 렌더링, 키보드 입력 처리 등의 요소를 활용하였다. 또한, 사운드 출력 및 게임 플레이 요소를 미로의 컨셉에 맞춰 조화롭게 구성하여 완성도를 높였다.

본 과제의 핵심은 SOR 화면 출력 및 미로 구현이다. 특히, SOR 화면에서 모델링 데이터를 저장하고 이를 미로에서 불러오는 방식의 구현이 중요하다고 판단하였다. 다양한 구현 방법이 있을 수 있으나, 본 과제에서는 SOR 화면에서 생성된 모델링 데이터를 저장한 뒤, 이를 미로에 바로 불러와 오브젝트를 배치하는 방식을 채택하였다. 이 방식은 데이터의 직관적인 관리와 활용이 용이하며, 과제의 기본 요구 사항을 충족하기에 적합하다고 보았다.

기본적인 기능 구현을 마친 후에는 추가기능 요소를 최대한으로 구현하는 것을 목표로 삼았다. 이를 위해 미로의 테마를 명확히 설정하고, 그에 맞는 요소를 설계하고 구현하였다. 본 과제에서 선택한 미로의 테마는 크리스마스 컨셉이다. 크리스마스를 연상시키는 트리, 눈사람, 트리 장식물 등을 활용하여 미로를 구성하였으며, 텍스처 매핑, 게임 플레이 요소, 사운드 출력 등을 테마에 부합하도록 제작하였다.

본론에서는 이러한 제작 과정을 흐름에 따라 상세히 설명하고, 작업 중 겪은 시행착오와 그에 대한 해결 방안을 소개한다.



## 2. 본론

### 2.1. 과제 개요 및 설계

#### 1. 프로그램의 개요와 설계 목표

본 프로젝트의 목표는 OpenGL을 활용해 크리스마스 테마의 미로 게임을 설계하고 구현하는 것이었다. 플레이어는 미로 안에서 크리스마스 장식물을 찾고, 모든 장식물을 수집하면 성공 이미지를 확인할 수 있다. 이를 위해 미로의 그래픽적 표현, 게임 오브젝트의 물리적 충돌 처리, 3D 환경 구성 및 사운드 효과를 통합적으로 구현하였다. 프로그램은 주로 OpenGL의 기본 기능과 C++의 객체지향 프로그래밍을 기반으로 개발되었다.



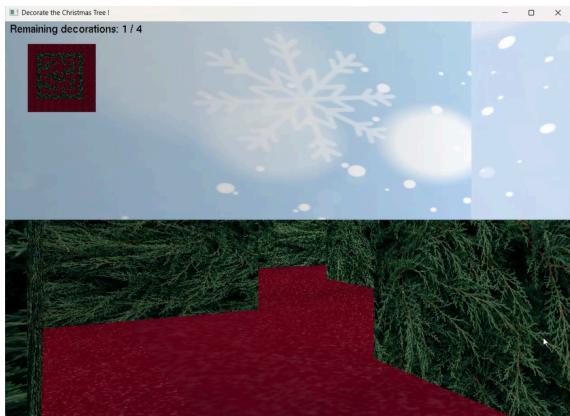
## 2. 프로그램 초기화

프로그램이 실행되면 가장 먼저 초기화 과정이 진행된다. 초기화는 OpenGL 렌더링 환경, 게임 오브젝트, 텍스처, 조명 설정 등으로 구성된다.

### 1. 텍스처 및 **Skybox** 설정

텍스처는 미로의 벽과 바닥을 보다 현실적으로 표현하기 위해 사용되었다.

`InitializeTexture` 함수를 통해 텍스처 파일을 로드하고 OpenGL 텍스처 객체를 생성하였다. 또한, **Skybox**는 미로의 하늘을 표현하기 위해 추가되었으며, `InitializeSkybox` 함수에서 텍스처가 큐브 형태로 렌더링되도록 구현하였다.



### 2. 조명 설정

조명은 미로와 오브젝트를 보다 입체적으로 표현하기 위해 필수적이다.

`InitializeLighting` 함수에서 OpenGL의 조명을 활성화하고, 조명 색상 및 위치를 설정하였다.

### 3. 오브젝트 초기화

미로에 배치된 오브젝트들은 `InitializeGameObjects` 함수를 통해 초기화된다.

`modelMaze` 배열에 저장된 데이터를 기반으로 각 오브젝트의 위치, 모델, 활성 상태 등을 `GameObject` 구조체로 저장하였다. 4개의 오브젝트 중에서 3개의 오브젝트(트리, 벌, 공)은 SOR 프로그램을 통해 사전에 저장된 `.dat` 파일을 불러와, 미로에 미리 배치시켜 놓았다.

눈사람 오브젝트의 경우, 앞선 SOR 프로그램을 실행시켜 x축 회전으로 생성된 점 정보를 실시간으로 `.dat` 파일을 불러오도록 하였다. 이는 시연 과정에서 직접 보여주기 위한 것이다.

Simple Modeling

tree.dat    snowman.dat

파일 편집 보기

```
VERTEX = 2812
-1.000000 200.000000 0.000000
-0.984808 200.000000 0.173648
-0.939693 200.000000 0.342020
-0.866025 200.000000 0.500000
-0.766044 200.000000 0.642788
-0.642788 200.000000 0.766044
-0.500000 200.000000 0.866025
-0.342020 200.000000 0.939693
-0.173648 200.000000 0.984808
0.000000 200.000000 1.000000
0.173648 200.000000 0.984808
0.342020 200.000000 0.939693
0.500000 200.000000 0.866025
0.642788 200.000000 0.766044
0.766044 200.000000 0.642788
0.866025 200.000000 0.500000
0.939693 200.000000 0.342020
0.984808 200.000000 0.173648
1.000000 200.000000 -0.000000
0.984808 200.000000 -0.173648
```

줄 1, 열 1 160,564자 100% Windows (CRLF) UTF-8

Simple Modeling

tree.dat    bell.dat    snowman.dat ball.dat

파일 편집 보기

```
VERTEX = 666
-2.000000 180.000000 0.000000
-1.969615 180.000000 0.347296
-1.879385 180.000000 0.684040
-1.732051 180.000000 1.000000
-1.532089 180.000000 1.285575
-1.285575 180.000000 1.532089
-1.000000 180.000000 1.732051
-0.684040 180.000000 1.879385
-0.347296 180.000000 1.969615
0.000000 180.000000 2.000000
0.347296 180.000000 1.969615
0.684040 180.000000 1.879385
1.000000 180.000000 1.732051
1.285575 180.000000 1.532089
1.532089 180.000000 1.285575
1.732051 180.000000 1.000000
1.879385 180.000000 0.684040
1.969615 180.000000 0.347297
2.000000 180.000000 -0.000000
1.969615 180.000000 -0.347296
```

줄 1, 열 1 34,303자 100% Windows (CRLF) UTF-8

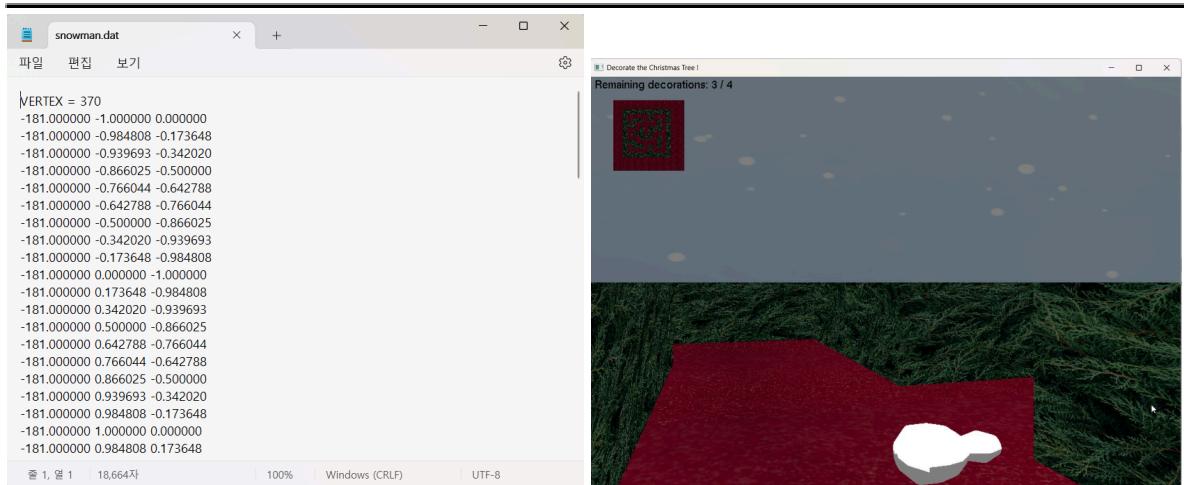
Simple Modeling

ball.dat    tree.dat    bell.dat    snowman.dat

파일 편집 보기

```
VERTEX = 259
1.000000 82.000000 0.000000
0.984808 82.000000 -0.173648
0.939693 82.000000 -0.342020
0.866025 82.000000 -0.500000
0.766044 82.000000 -0.642788
0.642788 82.000000 -0.766044
0.500000 82.000000 -0.866025
0.342020 82.000000 -0.939693
0.173648 82.000000 -0.984808
-0.000000 82.000000 -1.000000
-0.173648 82.000000 -0.984808
-0.342020 82.000000 -0.939693
-0.500000 82.000000 -0.866025
-0.642788 82.000000 -0.766044
-0.766044 82.000000 -0.642788
-0.866025 82.000000 -0.500000
-0.939693 82.000000 -0.342020
-0.984808 82.000000 -0.173648
-1.000000 82.000000 0.000000
-0.984808 82.000000 0.173648
```

줄 1, 열 1 12,425자 100% Windows (CRLF) UTF-8



### 3. OpenGL 렌더링 과정

본 프로그램의 렌더링 과정은 크게 두 가지 시점으로 나눈다.

#### 1. 쿼터뷰(Quarter View)

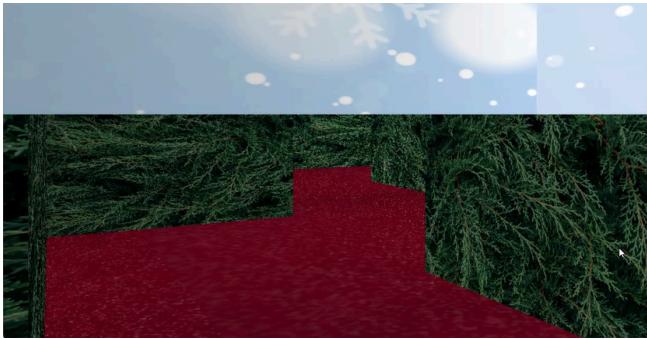
플레이어가 미로를 쉽게 파악할 수 있도록 상단에서 미로를 내려다보는 쿼터뷰가 구현되었다. `RenderQuarterView` 함수는 미로의 구조와 오브젝트를 상단에서 관찰할 수 있도록 설정된 카메라 시점을 기반으로 렌더링을 수행한다.



#### 2. 1인칭 뷰(First Person View)

플레이어가 실제로 미로 안을 이동하며 오브젝트를 찾는 시점이다.

`RenderFirstPersonView` 함수는 플레이어의 위치와 방향을 기반으로 카메라를 설정하여 미로를 탐험하는 경험을 제공한다.



이외에도 Skybox와 바닥 텍스처, 남아있는 오브젝트 개수 등을 화면에 렌더링하여 플레이어가 게임의 진행 상태를 쉽게 파악할 수 있도록 하였다.

#### 4. 충돌 처리 및 이벤트 구현

게임의 주요 기능 중 하나는 플레이어가 오브젝트와 상호작용할 수 있도록 충돌 처리를 구현하는 것이다.

##### 1. 벽과의 충돌

플레이어가 이동하려는 위치가 미로의 벽인지 확인하는 `CheckCollision` 함수가 이를 담당한다. 벽과 충돌 시 플레이어의 이동이 제한된다.

##### 2. 오브젝트와의 충돌

플레이어가 오브젝트와 충돌하면 `CheckObjectCollision` 함수가 호출되어 오브젝트가 비활성화된다. 오브젝트가 비활성화되면 사운드 효과가 재생되고, 남아있는 오브젝트의 개수가 화면에 업데이트된다.

##### 3. 성공 조건 확인

모든 오브젝트가 비활성화되었을 때, 성공 상태(`isSuccess`)로 전환된다. 성공 상태에서는 크리스마스 음악이 재생되며, 일정 시간 후 성공 이미지가 화면 전체에 표시된다. 이 과정은 `ShowSuccessImage` 및 `RenderSuccessImage` 함수에서 처리된다.

#### 5. 사운드 효과

게임의 몰입도를 높이기 위해 배경 음악과 효과음을 추가하였다.

- 배경 음악은 `BackgroundSound` 함수를 통해 게임이 실행되는 동안 재생되며, 플레이어가 오브젝트를 비활성화할 때 효과음(`ItemSound`)이 짧게 재생된다.
- 모든 오브젝트를 비활성화한 후에는 `PlayChristmasSound` 함수가 호출되어 크리스마스 테마 음악이 재생된다.

## 6. 프로그램 흐름 요약

프로그램은 다음과 같은 흐름으로 작동한다:

1. 초기화 단계에서 텍스처, 조명, Skybox, 오브젝트 등을 설정한다.
2. OpenGL의 메인 루프를 통해 화면에 미로와 오브젝트를 렌더링한다.
3. 플레이어의 이동 및 키보드 입력이 반영되며, 충돌 처리를 통해 게임 상태가 업데이트된다.
4. 모든 오브젝트를 비활성화하면 성공 상태로 전환되며, 성공 이미지와 음악이 출력된다.

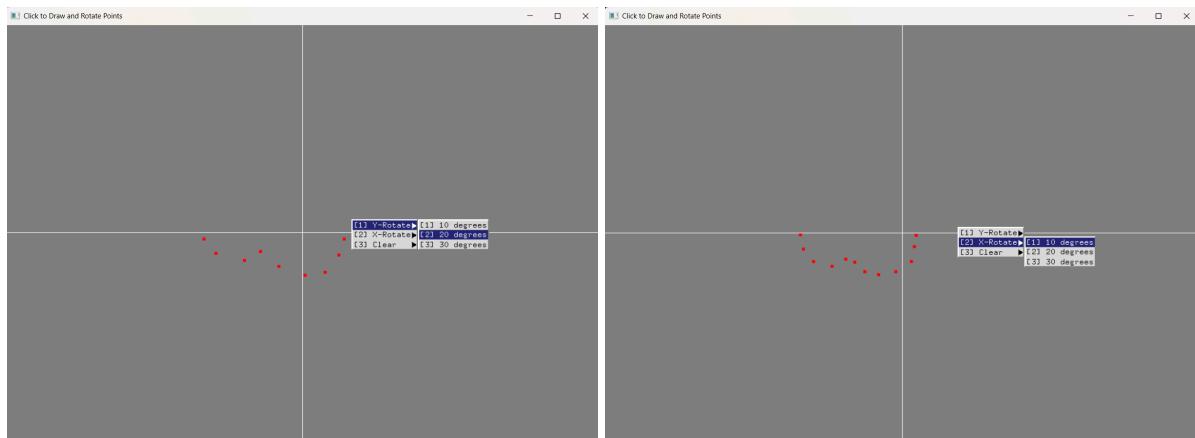
본 프로그램은 OpenGL의 기본 기능과 텍스처, 충돌 처리, 사운드 효과를 활용하여 게임 플레이 경험을 구현하였다. 특히, Skybox와 크리스마스 테마 요소는 미로의 몰입도를 높이는 데 큰 기여를 하였다.

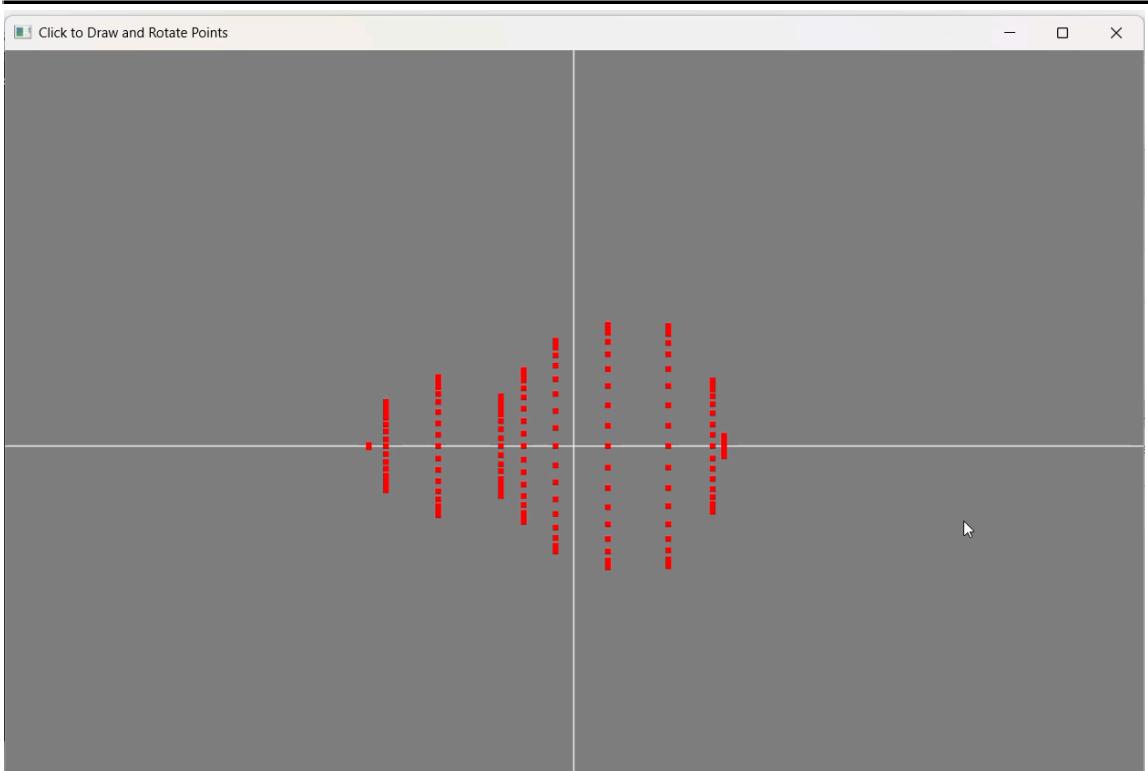
## 2.2. 주요기능

### 1. SOR 파일 화면 실행 및 회전 변환 프로그램

이 코드는 OpenGL과 GLUT를 활용해 점을 입력받아 화면에 표시하고, 입력된 점을 기준으로 X축 및 Y축 회전 변환을 통해 3D 모델을 생성 및 저장하는 프로그램이다. 주요 기능은 점 입력, 회전 변환, 면 생성, 모델 저장, 화면 렌더링으로 구성된다.

마우스 우클릭 시 나타나는 메뉴창을 통해, x축 및 y축 회전 시 각각 10, 20 30도 회전을 선택할 수 있으며, 클릭 시 기존 입력된 점들의 회전된 점들이 모두 표현된다. 또한 이 점들의 위치값이 .dat파일로 바로 저장된다.



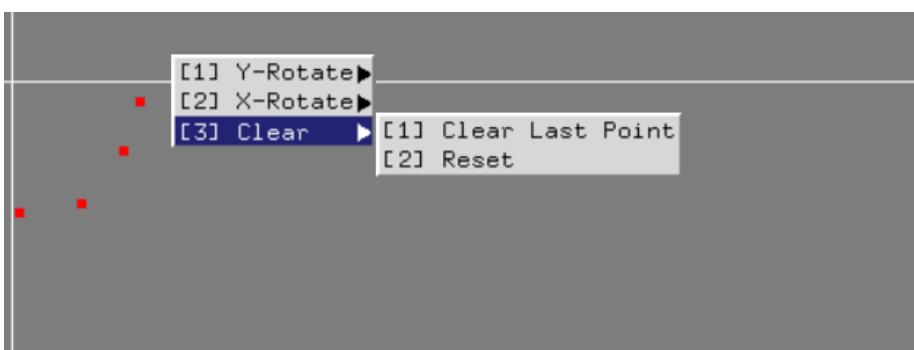


```

snowman.dat      snowman.dat
파일 편집 보기   파일 편집 보기
VERTEX = 370      FACE = 648
-181.000000 -1.000000 0.000000
-181.000000 -0.98408 -0.173648
-181.000000 -0.939693 -0.342020
-181.000000 -0.866025 -0.500000
-181.000000 -0.766044 -0.642788
-181.000000 -0.642788 -0.766044
-181.000000 -0.500000 -0.866025
-181.000000 -0.342020 -0.939693
-181.000000 -0.173648 -0.98408
-181.000000 0.000000 -1.000000
-181.000000 0.173648 -0.98408
-181.000000 0.342020 -0.939693
-181.000000 0.500000 -0.866025
-181.000000 0.642788 -0.766044
-181.000000 0.766044 -0.642788
-181.000000 0.866025 -0.500000
-181.000000 0.939693 -0.342020
-181.000000 0.98408 -0.173648
-181.000000 1.000000 0.000000
-181.000000 0.98408 0.173648
  줄 1, 열 1 18,664자  100% Windows (CRLF) UTF-8
  줄 1, 열 1 18,664자  100% Windows (CRLF) UTF-8

```

Clear 메뉴 내에 하위 메뉴를 만들어, 직전에 찍은 한 점만 지울 것인지, 모든 점들을 지울 것인지를 선택할 수 있다.



## 1.1. 클래스 및 전역 변수 정의

`xPoint3D` 클래스는 3D 좌표( $x, y, z$ )와 가중치( $w$ )를 표현하며, 점 데이터를 관리한다. `arInputPoints`는 사용자가 입력한 점들을, `arRotPoints`는 회전 변환 결과를, `arFaces`는 3D 모델의 면(삼각형)을 정점 인덱스로 저장한다. 윈도우 크기와 점 개수(`pnum`), 면 개수(`fnum`)를 전역 변수로 관리한다.

```
class xPoint3D
{
public:
    float x, y, z, w;
    xPoint3D() { x = y = z = 0; w = 1; };
    xPoint3D(float x, float y, float z, float w = 1.0f) : x(x), y(y), z(z), w(w) {};
};

vector<xPoint3D> arInputPoints;
vector<xPoint3D> arRotPoints;
vector<vector<int>> arFaces;
GLsizei winWidth = 1000, winHeight = 700;
int pnum = 0;
int fnum = 0;
```

## 1.2. OpenGL 초기화 및 기본 설정

초기화 함수 `init`는 OpenGL 환경을 설정하며, 배경색을 회색으로 지정(`glClearColor`)하고 직교 투영(`glOrtho`)을 사용해 중심을 기준으로 좌표계를 설정한다.

```
void init(void)
{
    glClearColor(0.5, 0.5, 0.5, 1.0);
    glMatrixMode(GL_PROJECTION);
    glOrtho(-winWidth / 2, winWidth / 2, -winHeight / 2, winHeight / 2, -500.0, 500.0);
}
```

## 1.3. 점 입력 및 화면 표시

`mouseClick` 함수는 마우스 좌표를 중심 기준으로 변환하여 입력된 점을 저장하며, `myDisplay` 함수는  $x$ 축과  $y$ 축을 그린 뒤 입력된 점(`arInputPoints`)과 회전된 점(`arRotPoints`)을 화면에 표시한다. 입력된 점은 `GL_POINTS`로 렌더링되며, 화면 갱신은 `glutPostRedisplay`로 처리된다.

```
void mouseClicked(GLint button, GLint action, GLint xMouse, GLint yMouse)
{
    if (button == GLUT_LEFT_BUTTON && action == GLUT_DOWN)
    {
        xPoint3D pt;
        pt.x = xMouse - winWidth / 2;
        pt.y = (winHeight - yMouse) - winHeight / 2;
        pt.z = 0;

        arInputPoints.push_back(pt);
        glutPostRedisplay();
    }
}
```

입력된 점과 회전된 점을 화면에 렌더링 한다.

```
void myDisplay(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glColor3f(1.0, 1.0, 1.0);
    glBegin(GL_LINES);
    glVertex3f(-winWidth / 2, 0, 0);
    glVertex3f(winWidth / 2, 0, 0);
    glVertex3f(0, -winHeight / 2, 0);
    glVertex3f(0, winHeight / 2, 0);
    glEnd();

    plotPoints();
    glFlush();
}
```

#### 1.4. 회전 변환 및 면 생성

xRotation과 yRotation 함수는 입력된 점들을 기준으로 X축과 Y축 회전을 수행한다. 각도(fRotAngle)를 라디안으로 변환하여 360도 범위에서 일정 간격으로 회전하며, 결과를 arRotPoints에 저장한다. generateFaces 함수는 각 점의 인덱스를 사용해 링 형태의 점을 연결하여 삼각형 면을 생성한다.

```
void myXRotationMenu(int angleID)
{
    if (angleID == 1)
        xRotation(10.0f);
    else if (angleID == 2)
        xRotation(20.0f);
    else if (angleID == 3)
        xRotation(30.0f);

    saveModel();
    glutPostRedisplay();

    this_thread::sleep_for(std::chrono::milliseconds(1000));
    glutMainLoopEvent();

    closeWindow();

    glutLeaveMainLoop();

    system("C:\Projects\CG_practice\Debug\CG_practice.exe");
    exit(0);
}

void myYRotationMenu(int angleID)
{
    if (angleID == 1)
        yRotation(10.0f);
    else if (angleID == 2)
        yRotation(20.0f);
    else if (angleID == 3)
        yRotation(30.0f);

    saveModel();
    glutPostRedisplay();

    this_thread::sleep_for(std::chrono::milliseconds(1000));
    glutMainLoopEvent();

    closeWindow();

    glutLeaveMainLoop();

    system("C:\Projects\CG_practice\Debug\CG_practice.exe");
    exit(0);
}
```

링 형태로 연결된 점들을 삼각형 면으로 연결하여 3D 모델의 면 데이터를 생성한다.

```

void generateFaces(int segments)
{
    arFaces.clear();
    int pointsPerRing = segments;
    int totalRings = pnum;

    for (int i = 0; i < totalRings - 1; ++i)
    {
        int baseIndex = i * pointsPerRing;
        int nextBaseIndex = (i + 1) * pointsPerRing;

        for (int j = 0; j < pointsPerRing; ++j)
        {
            int current = baseIndex + j;
            int next = baseIndex + (j + 1) % pointsPerRing;
            int nextRing = nextBaseIndex + j;
            int nextRingNext = nextBaseIndex + (j + 1) % pointsPerRing;

            arFaces.push_back({ current, next, nextRing });
            arFaces.push_back({ next, nextRingNext, nextRing });
        }
    }
}

```

## 1.5. 모델 저장

`saveModel` 함수는 `snowman.dat` 파일에 정점 데이터와 면 정보를 저장한다.

파일에는 정점 개수, 정점 좌표, 면 정보가 순차적으로 기록되며, 이는 3D 모델 데이터를 직관적으로 관리할 수 있게 한다.

```

void saveModel()
{
    cout << "Saving model to snowman.dat...\\n";

    FILE* fout = fopen("c:\\\\data\\\\snowman.dat", "w");
    if (!fout)
    {
        cerr << "Failed to open file for writing. Check file path and permissions.\\n";
        return;
    }

    pnum = static_cast<int>(arInputPoints.size() + arRotPoints.size());
    fprintf(fout, "VERTEX = %d\\n", pnum);
    for (const auto& pt : arRotPoints)
    {
        fprintf(fout, "%f %f %f\\n", pt.x, pt.y, pt.z);
    }

    fprintf(fout, "FACE = %d\\n", static_cast<int>(arFaces.size()));
    for (const auto& face : arFaces)
    {
        fprintf(fout, "%d %d %d\\n", face[0], face[1], face[2]);
    }

    fclose(fout);
    cout << "Model saved to snowman.dat\\n";
}

```

---

이 과정은 입력 → 변환 → 면 생성 → 렌더링 및 저장의 과정을 통해 3D 모델을 생성하는 과정을 구현한다. OpenGL의 기본 기능을 활용해 확장성과 학습용으로 적합한 구조를 제공하며, 모델링의 기초 과정이다.

## 2. 미로 구현

미로 게임을 만들기 위한 주요기능은 미로 구현, 플레이어 이동 및 카메라 제어, 오브젝트 배치, 화면 렌더링으로 구성된다. 이 기능들을 통해 기본적인 미로 모델링을 완성할 수 있다.

### 2.1. 미로 데이터 정의

`maze[11][11]` 배열을 통해 1은 벽, 0은 이동 가능한 공간을 나타낸다.

```
int maze[11][11] = {  
    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},  
    {1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1},  
    {1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1},  
    {1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1},  
    {1, 0, 1, 0, 1, 1, 0, 1, 0, 1},  
    {1, 0, 0, 0, 0, 1, 0, 0, 0, 1},  
    {1, 1, 1, 0, 1, 1, 0, 1, 1, 1},  
    {1, 0, 0, 0, 1, 0, 0, 0, 0, 1},  
    {1, 0, 1, 1, 1, 0, 1, 1, 0, 1},  
    {1, 0, 0, 0, 0, 0, 0, 0, 0, 1},  
    {1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
```

---

## 2.2. 미로 렌더링 함수

1인칭 뷰와 쿼터뷰를 만들었으며 `gluLookAt`을 이용하여 방향을 설정했다. `gluPerspective`로는 60도의 FOV와 1:1화면 비율을 설정했다. `DrawFloor`를 통해 미로의 바닥을 렌더링하고 2중 `for`문을 사용해 `maze` 배열을 순회하며, 값이 1인 경우 `DrawCube`를 호출해 벽을 렌더링하였다.

```
void RenderQuarterView() {
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(45.0, 1.0, 0.1, 100.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0.0, 20.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0);

    DrawFloor(10.0f, 0.1f);

    int maze[11][11] = {
        {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
        {1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1},
        {1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1},
        {1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1},
        {1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1},
        {1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1},
        {1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1},
        {1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1},
        {1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1},
        {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
        {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
    };

    for (int i = 0; i < 11; i++) {
        for (int j = 0; j < 11; j++) {
            if (maze[i][j] == 1) {
                DrawCube(i * 0.6f - 3.3f, 0.0f, j * 0.6f - 3.3f);
            }
        }
    }
}
```

## 2.3 큐브 디스플레이 리스트 생성

8개의 정점(MyVertices)과 6개의 면(MyVertexList)으로 구성하여 각 면에 텍스처 맵핑을 하였다.

```
void MakeCubePlayList() {
    MyListID = glGenLists(1);
    glNewList(MyListID, GL_COMPILE);
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(3, GL_FLOAT, 0, MyVertices);

    glBindTexture(GL_TEXTURE_2D, mazeTexture);
    glEnable(GL_TEXTURE_2D);

    glBegin(GL_QUADS);
    for (GLint i = 0; i < 6; i++) {
        for (int j = 0; j < 4; j++) {
            int index = MyVertexList[i * 4 + j];
            glTexCoord2f((j == 1 || j == 2), (j == 2 || j == 3));
            glVertex3fv(MyVertices[index]);
        }
    }
    glEnd();

    glDisable(GL_TEXTURE_2D);
    glEndList();
}
```

## 2.4. 미로 충돌 감지

플레이어의 실시간 ( $x$ ,  $z$ ) 좌표를 maze 배열의 ( $mazeX$ ,  $mazeZ$ ) 좌표로 변환하여 배열 값이 1인 경우 충돌로 간주하였다.

```
bool CheckCollision(float newX, float newZ) {
    int mazeX = static_cast<int>(round((newX + 3.3f) / 0.6f));
    int mazeZ = static_cast<int>(round((newZ + 3.3f) / 0.6f));

    if (mazeX < 0 || mazeX >= 11 || mazeZ < 0 || mazeZ >= 11) {
        return true;
    }

    return maze[mazeX][mazeZ] == 1;
}
```

미로 구현 코드를 통해 미로의 구조를 정의하고 이를 화면에 렌더링하여 플레이어의 이동 및 충돌 감지를 처리하는 과정을 통해서 기본적인 미로 게임의 토대를 완성할 수 있게 되었다.

### 3. 플레이어의 이동 및 카메라 제어

플레이어가 자유롭게 미로를 탐험하고 시점을 조작할 수 있도록 한다. 게임의 기본적인 상호작용이다.

#### 3.1. 플레이어 위치와 방향

`playerX`, `playerY`는 초기좌표로 설정된다. `playerY`는 고정된 높이로 설정된다. 이 세 값은 플레이어의 위치를 의미한다. `playerYaw`은 카메라의 좌우 회전값, `cameraSpeed`는 이동 속도를 의미한다.

```
21
22     float playerX = -3.3f + 1 * 0.6f, playerY = 0.3f, playerZ = -3.3f + 1 * 0.6f;
23     float playerYaw = 0.0f;
24     float cameraSpeed = 0.1f;
```

#### 3.2. 키보드 입력 처리

`GLUT_KEY_UP`은 앞으로, `GLUT_KEY_DOWN`은 뒤로 이동처리를 하며 `GLUT_KEY_LEFT`은 왼쪽으로, `GLUT_KEY_RIGHT`은 오른쪽으로 회전 처리를 한다. `CheckCollision`을 통해 충돌체크 후 충돌 후에는 `CheckObjectCollision` 함수가 호출된다.

```
703
704     void ProcessSpecialKeys(int key, int x, int y) {
705         float newX = playerX;
706         float newZ = playerZ;
707
708         switch (key) {
709             case GLUT_KEY_UP:
710                 newX += cameraSpeed * cos(playerYaw);
711                 newZ += cameraSpeed * sin(playerYaw);
712                 break;
713             case GLUT_KEY_DOWN:
714                 newX -= cameraSpeed * cos(playerYaw);
715                 newZ -= cameraSpeed * sin(playerYaw);
716                 break;
717             case GLUT_KEY_LEFT:
718                 playerYaw -= 0.1f;
719                 glutPostRedisplay();
720                 return;
721             case GLUT_KEY_RIGHT:
722                 playerYaw += 0.1f;
723                 glutPostRedisplay();
724                 return;
725         }
726     }
```

#### 3.3. 마우스 입력 처리

마우스의 X축 움직임에 따라 `playerYaw`가 변경된다. `glutWarpPointer`를 사용해 마우스를 화면 중앙으로 고정하여 시점이 일관되게 유지된다.

```

697
698     void ProcessMouseMove(int x, int y) {
699         float sensitivity = 0.005f;
700         playerYaw += (x - winWidth / 2) * sensitivity;
701         glutWarpPointer(winWidth / 2, winHeight / 2);
702     }
703

```

### 3.4. 1인칭 뷰 및 스카이뷰

플레이어 이동 및 카메라 제어는 1인칭 시점과 퀘터뷰를 제공하여, 키보드와 마우스 입력에 따라 유동적으로 반응한다. 총돌 감지와 연계하여 플레이어가 물리적인 제약을 받는 현실감 있는 환경을 구현하며, 다양한 시점 전환을 통해 게임플레이의 편의성을 높일 수 있다.

```

514
515     void RenderQuarterView() {
516         glMatrixMode(GL_PROJECTION);
517         glLoadIdentity();
518         gluPerspective(45.0, 1.0, 0.1, 100.0);
519         glMatrixMode(GL_MODELVIEW);
520         glLoadIdentity();
521
522         gluLookAt(0.0, 20.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0);
523
524         DrawFloor(10.0f, 0.1f);
525
526         int maze[11][11] = {
527             {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
528             {1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1},
529             {1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1},
530             {1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1},
531             {1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1},
532             {1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1},
533             {1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1},
534             {1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1},
535             {1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1},
536             {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
537             {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
538         };
539

```

```

540
541     for (int i = 0; i < 11; i++) {
542         for (int j = 0; j < 11; j++) {
543             if (maze[i][j] == 1) {
544                 DrawCube(i * 0.6f - 3.3f, 0.0f, j * 0.6f - 3.3f);
545             }
546         }
547     }
548
549

```

```

550     void RenderFirstPersonView() {
551         glMatrixMode(GL_PROJECTION);
552         glLoadIdentity();
553         gluPerspective(60.0, 1.0, 0.1, 100.0);
554         glMatrixMode(GL_MODELVIEW);
555         glLoadIdentity();
556
557         float lookX = playerX + cos(playerYaw);
558         float lookZ = playerZ + sin(playerYaw);
559
560         gluLookAt(playerX, playerY, playerZ, lookX, playerY, lookZ, 0.0f, 1.0f, 0.0f);
561
562         DrawFloor(10.0f, 0.1f);
563
564         int maze[11][11] = {
565             {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1},
566             {1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1},
567             {1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1},
568             {1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1},
569             {1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1},
570             {1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1},
571             {1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1},
572             {1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1},
573             {1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1},
574             {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1},
575             {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}
576         };
577
577
578         for (int i = 0; i < 11; i++) {
579             for (int j = 0; j < 11; j++) {
580                 if (maze[i][j] == 1) {
581                     DrawCube(i * 0.6f - 3.3f, 0.0f, j * 0.6f - 3.3f);
582                 }
583             }
584         }
585     }

```

## 4. 오브젝트 배치

### 4.1. 미로 데이터와 오브젝트 초기화

2D 배열로 미로의 각 좌표에 배치될 오브젝트를 나타낸다.

```

int modelMaze[11][11] = {
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0},
    {0, 1, 0, 0, 0, 0, 0, 0, 2, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 4, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
};

```

## 4.2. 오브젝트 초기화 함수

`activeObjectCount`를 0으로 초기화하여 활성화된 오브젝트를 초기화한다. `x, y, z` 좌표는 `modelMaze`의 인덱스를 기준으로 변환된다. `modelMaze[i][j] == 3`(공)일 경우에는 `vx, vz`: 속도를 임의의 값으로 초기화하여 동적인 움직임을 구현하고 `ax, az`: 가속도는 초기에는 0으로 설정한다. `gameObjects` 벡터에 오브젝트를 추가하여 성화된 오브젝트의 개수를 증가시켜 충돌 감지와 성공 조건 체크에 활용한다.

```
void InitializeGameObjects() {
    activeObjectCount = 0;

    for (int i = 0; i < 11; i++) {
        for (int j = 0; j < 11; j++) {
            if (modelMaze[i][j] != 0) {
                float x = i * 0.6f - 3.3f;
                float z = j * 0.6f - 3.3f;
                float y = 0.1f;

                GameObject obj = { x, y, z, 0.0f, 0.0f, 0.0f, 0.0f, 0.0f, modelMaze[i][j], true };

                if (modelMaze[i][j] == 3) {
                    obj.vx = ((rand() % 100) / 100.0f - 0.5f) * 0.2f;
                    obj.vz = ((rand() % 100) / 100.0f - 0.5f) * 0.2f;
                    obj.ax = 0.0f;
                    obj.az = 0.0f;
                }

                gameObjects.push_back(obj);
                activeObjectCount++;
            }
        }
    }
}
```

## 4.3. 오브젝트 렌더링

`gameObjects` 벡터에 저장된 오브젝트를 반복하여 렌더링한다. `obj.active == true`인 경우에만 해당 오브젝트를 렌더링 한다. `modelIndex`에 따라 적절한 모델 파일(`tree.dat`, `bell.dat` 등)을 읽어와 해당 좌표(`x, y, z`)에 배치한다.

```
void RenderObjects() {
    for (const auto& obj : gameObjects) {
        if (obj.active) {
            ReadModel(obj.modelIndex, obj.x, obj.y, obj.z);
        }
    }
}
```

## 4.4. 충돌 후 오브젝트 비활성화

플레이어와 오브젝트의 위치 간 거리를 계산하여 충돌 여부를 판단한다. 충돌한 경우에는 해당 오브젝트를 비활성화(`obj.active = false`) 처리하여 활성화된 오브젝트 개수를 감소시킨다.

```

void CheckObjectCollision() {
    for (auto& obj : gameObjects) {
        if (obj.active) {
            float distance = sqrt(pow(playerX - obj.x, 2) + pow(playerZ - obj.z, 2));
            if (distance < 0.3f) {
                obj.active = false;
                activeObjectCount--;
                ItemSound();
                std::cout << "Object deactivated. Remaining active objects: " << activeObjectCount << "\n";
            }
        }
    }
}

```

오브젝트 배치는 미로의 초기 데이터를 기반으로 동작하며, 각 오브젝트의 위치, 특수 동작(공의 움직임 등), 활성화 상태를 관리한다. 이를 통해 플레이어와의 상호작용을 구현하고, 성공 조건을 확인하는 중요한 역할을 수행한다.

## 2.3. 추가기능

기본적인 미로 구현을 마친 뒤 미로의 컨셉과 게임 요소를 추가하기 위해서 추가기능을 포함하였다.

텍스처 매핑, 사운드 출력, 오브젝트와의 충돌 처리를 구현하여 게임 플레이적 요소와 미로의 컨셉에 맞는 그래픽을 포함할 수 있게 되었다.

### 1. 텍스처 매핑

텍스처 매핑은 수업 시간에 배웠던 **bmp** 파일 변환을 통해서 구현하였다. 크리스마스 컨셉을 구현하기 위해서 미로 큐브의 벽면에는 트리나무 이미지를, 바닥 면에는 레드카펫 이미지를 매핑하여 크리스마스 분위기를 살려내었다. 텍스처 맵핑을 위해서는 이미지 파일 읽기-> 텍스처 생성 및 초기화 -> 텍스처 적용-> 렌더링에 텍스처 사용 순으로 단계가 구성된다.

#### 1.1. 이미지 파일 읽기

이 코드는 **bmp** 파일을 읽고 픽셀데이터를 추출하여 OpenGL에 사용할 수 있도록 변환하는 과정이다.

```

unsigned char* Read_BmpImage(const char name[], int* width, int* height, int* components) {
    FILE* BMPfile;
    errno_t err;
    GLubyte garbage;
    long size;
    int start_point, x;
    GLubyte temp[3];
    GLubyte start[4], w[4], h[4];
    unsigned char* read_image;

    err = fopen_s(&BMPfile, name, "rb");
    if (err != 0 || BMPfile == NULL) {
        cerr << "Error opening file: " << name << endl;
        return nullptr;
    }

    for (x = 0; x < 10; x++) {
        fread(&garbage, 1, 1, BMPfile);
    }

    fread(&start[0], 1, 4, BMPfile);

    for (x = 0; x < 4; x++) {
        fread(&garbage, 1, 1, BMPfile);
    }

    fread(&w[0], 1, 4, BMPfile);
    fread(&h[0], 1, 4, BMPfile);

    (*width) = (w[0] + 256 * w[1] + 256 * 256 * w[2] + 256 * 256 * 256 * w[3]);
    (*height) = (h[0] + 256 * h[1] + 256 * 256 * h[2] + 256 * 256 * 256 * h[3]);
    size = (*width) * (*height);
    start_point = (start[0] + 256 * start[1] + 256 * 256 * start[2] + 256 * 256 * 256 * start[3]);

    read_image = (unsigned char*)malloc(size * 3);

    for (x = 0; x < (start_point - 26); x++) {
        fread(&garbage, 1, 1, BMPfile);
    }

    for (x = 0; x < (size * 3); x = x + 3) {
        fread(&temp[0], 1, 3, BMPfile);
        read_image[x] = temp[2];
        read_image[x + 1] = temp[1];
        read_image[x + 2] = temp[0];
    }
    fclose(BMPfile);
    return read_image;
}

```

## 1.2. 텍스처 초기화

`glGenTexture`로 텍스처 객체를 생성하고 `glTexImage2D`로 텍스처 이미지를 OpenGL에 업로드한다. 그 후 `glTexParameter`으로 텍스처 필터링 및 반복설정을 통해 렌더링에 사용할 준비를 한다.

```

void InitializeTexture(GLuint* texture, const std::string& fileName) {
    unsigned char* imageData;
    int width, height, depth;
    imageData = Read_BmpImage(fileName.c_str(), &width, &height, &depth);

    if (imageData == nullptr) {
        cerr << "Error: Failed to load texture: " << fileName << endl;
        return;
    }

    glGenTextures(1, texture);
    glBindTexture(GL_TEXTURE_2D, *texture);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, imageData);

    free(imageData);
}

```

### 1.3. 텍스처 적용

초기화된 텍스처를 `glTexCoord2f`를 통해 텍수처 좌표를 정점에 매핑하고 `glVertex3v`를 통해 큐브 정점 데이터를 사용하여 텍스처를 매핑한다.

```
void MakeCubePlayList() {
    MyListID = glGenLists(1);
    glNewList(MyListID, GL_COMPILE);
    glEnableClientState(GL_VERTEX_ARRAY);
    glVertexPointer(3, GL_FLOAT, 0, MyVertices);

    glBindTexture(GL_TEXTURE_2D, mazeTexture);
    glEnable(GL_TEXTURE_2D);

    glBegin(GL_QUADS);
    for (GLint i = 0; i < 6; i++) {
        for (int j = 0; j < 4; j++) {
            int index = MyVertexList[i * 4 + j];
            glTexCoord2f((j == 1 || j == 2), (j == 2 || j == 3));
            glVertex3fv(MyVertices[index]);
        }
    }
    glEnd();

    glDisable(GL_TEXTURE_2D);
    glEndList();
}
```

이 과정을 완료하면 미로의 벽과 바닥에는 트리와 레드카펫이 깔리게되고 크리스마스라는 컨셉의 시각적 몰입감이 향상될 수 있다.

## 2. 사운드 효과

사운드 효과는 게임의 몰입감을 높이기 위해 중요한 요소로, 배경 음악과 효과음을 통해 플레이어에게 청각적인 즐거움을 제공한다. 본 과제에서는 OpenGL과 함께 Windows의 `PlaySound API`를 사용하여 간단한 사운드 효과를 구현하였다. 사운드 구현은 다음과 같은 단계로 구성된다.

### 2.1. 사운드 파일 준비 및 경로 설정

게임에서 재생할 사운드 파일은 `*.wav` 형식으로 준비한다. 크리스마스 테마를 강조하기 위해 다음과 같은 사운드를 선택하였다

- 배경 음악: 크리스마스 분위기를 위한 배경 음악 (`background.wav`)
- 효과음: 오브젝트와의 상호작용에서 재생되는 효과음 (`item.wav`)
- 성공 음악: 모든 목표를 달성했을 때 재생되는 음악 (`success.wav`)

---

이 사운드 파일들은 실행 파일 경로에 위치해야 한다.

## 2.2. 사운드 재생 함수 정의

사운드 재생은 Windows API의 **PlaySound** 함수를 활용한다. 사운드 파일 경로를 전달하고, 재생 모드를 설정하여 필요한 시점에서 적절한 사운드를 재생한다. **BackgroundSound**은 게임 실행 시 기본적으로 재생되는 배경음이다. **ItemSound**은 오브젝트와 달아 오브젝트를 수집할 때 재생되는 효과음이다. **PlayChristmasSound**은 모든 오브젝트를 수집하여 크리스마스 트리 완성을 축하하며 게임 클리어 화면을 보여주는 배경음이다. 사운드들을 활용하여 크리스마스 분위기를 더 잘 표현할 수 있었다.

```

12 | #include <windows.h>

472
473     void PlayChristmasSound() {
474         PlaySound(TEXT("JingleBells.wav"), NULL, SND_ASYNC | SND_LOOP);
475     }
476
477     void BackgroundSound() {
478         PlaySound(TEXT("Christmas Special 4 (Instrumental).wav"), NULL, SND_ASYNC | SND_LOOP);
479     }
480
481     void ItemSound() {
482         std::thread t[] {
483             PlaySound(TEXT("Item2A.wav"), NULL, SND_SYNC);
484
485             if (activeObjectCount != 0)
486             {
487                 BackgroundSound();
488             }
489         }.detach();
490     }
491

```

- **SND\_FILENAME**: 사운드 파일의 경로를 지정.
- **SND\_ASYNC**: 비동기 방식으로 재생하여 게임의 진행을 멈추지 않음.
- **SND\_LOOP**: 배경 음악을 반복 재생.

## 2.3. 배경 음악 재생

배경 음악은 프로그램이 시작될 때 실행되어야 하므로, main 함수에서 호출된다.

```

783     int main(int argc, char** argv) {
784         glutInit(&argc, argv);
785         glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH);
786         glutInitWindowPosition(270, 45);
787         glutInitWindowSize(winWidth, winHeight);
788         glutCreateWindow("Decorate the Christmas Tree !");
789
790         glClearColor(1.0, 1.0, 1.0, 1.0);
791         glEnable(GL_DEPTH_TEST);
792
793         InitializeTexture(&mazeTexture, textureFileName);
794         InitializeTexture(&floorTexture, floorTextureFileName);
795         InitializeLighting();
796         MakeCubePlayList();
797         InitializeGameObjects();
798         InitializeSuccessTexture();
799         InitializeSkybox();
800
801         glutDisplayFunc(MyDisplay);
802         glutSpecialFunc(ProcessSpecialKeys);
803
804         BackgroundSound();
805
806         glutMainLoop();
807         return 0;
808     }

```

## 2.4. 효과음 재생

효과음은 플레이어가 오브젝트와 상호작용하거나 충돌했을 때 호출된다. 이를 위해 충돌 처리 함수와 연계한다. 남은 오브젝트 개수가 하나씩 감소할 때마다 효과음이 재생된다.

```

491
492     void CheckObjectCollision() {
493         for (auto& obj : gameObjects) {
494             if (obj.active) {
495                 float distance = sqrt(pow(playerX - obj.x, 2) + pow(playerZ - obj.z, 2));
496                 if (distance < 0.3f) {
497                     obj.active = false;
498                     activeObjectCount--;
499                     ItemSound();
500                     std::cout << "Object deactivated. Remaining active objects: " << activeObjectCount << "\n";
501                 }
502             }
503         }
504     }

```

## 2.5. 성공 음악 재생

모든 오브젝트를 비활성화한 뒤, 성공 조건을 충족했을 때 성공 음악이 재생된다. 이 음악은 기존 배경 음악을 멈추고 실행된다.

```

501
502     if (activeObjectCount == 0 && !isSuccess) {
503         isSuccess = true;
504         std::cout << "Success! All objects are deactivated.\n";
505         PlaySound(NULL, NULL, 0);
506         PlayChristmasSound();
507         glutTimerFunc(successDelay, ShowSuccessImage, 0);
508     }
509 }
510 }
511 }
512 }
513 }

```

사운드 효과는 단순한 기능 구현을 넘어, 크리스마스 테마를 강조하고 게임의 몰입감을 높이는 데 큰 기여를 하였다. 배경 음악, 효과음, 성공 음악을 조화롭게 연계하여 게임 플레이 경험을 풍성하게 만들었다.

### 3. 충돌 체크

충돌 체크 기능은 플레이어의 맵, 오브젝트와 플레이어 간의 상호작용, 충돌 후 성공 이벤트 처리리를 위해 구현하였다.

플레이어가 미로의 맵을 벗어나지 않도록 제한하고 오브젝트와 플레이어의 충돌 여부를 감지해 모든 오브젝트와 충돌 완료시에는 성공 조건을 모두 충족하게 되어 축하 이벤트 시스템을 시행시켰다.

#### 3.1. 벽과의 충돌 체크

플레이어의 newX, newY를 미로 좌표계로 변환 한 후 미로 범위를 벗어난 경우에 충돌로 처리한다. maze 배열에서 위치 값이 1인지 확인하고 1이면 충돌, 0이면 이동 가능하게 만든다.

```
bool CheckCollision(float newX, float newY) {
    int mazeX = static_cast<int>(round((newX + 3.3f) / 0.6f));
    int mazeZ = static_cast<int>(round((newY + 3.3f) / 0.6f));

    if (mazeX < 0 || mazeX >= 11 || mazeZ < 0 || mazeZ >= 11) {
        return true;
    }

    return maze[mazeX][mazeZ] == 1;
}
```

#### 3.2. 오브젝트와 충돌체크

active== true인 오브젝트만 확인하여 플레이어와의 유클리드 거리를 계산한다. 거리가 0.3f이하면 충돌로 간주한다. 충돌시에는 해당 오브젝트를 active=false하여 활성화 된 오브젝트의 개수를 감소시킨다.

```

void CheckObjectCollision() {
    for (auto& obj : gameObjects) {
        if (obj.active) {
            float distance = sqrt(pow(playerX - obj.x, 2) + pow(playerZ - obj.z, 2));
            if (distance < 0.3f) {
                obj.active = false;
                activeObjectCount--;
                ItemSound();
                std::cout << "Object deactivated. Remaining active objects: " << activeObjectCount << "\n";
            }
            if (activeObjectCount == 0 && !isSuccess) {
                isSuccess = true;
                std::cout << "Success! All objects are deactivated.\n";
                PlaySound(NULL, NULL, 0);
                PlayChristmasSound();
                glutTimerFunc(successDelay, ShowSuccessImage, 0);
            }
        }
    }
}

```

### 3.3. 충돌 후 이벤트 처리

`ProcessSpecialKeys` 함수 내에서 플레이어의 이동방향에 따라 플레이어의 잠재적인 새 위치를 계산한다. 벽 충돌 체크는 충돌시 위치 갱신을 하지 않고 오브젝트 충돌 체크는 충돌 시 오브젝트 이벤트 처리를 한다. 그 후 화면 갱신을 한다.

```

726
727     if (!CheckCollision(newX, newZ)) {
728         playerX = newX;
729         playerZ = newZ;
730     }
731
732     CheckObjectCollision();
733
734     glutPostRedisplay();
735 }

```

충돌 체크 기능을 통해 게임 플레이의 기본 요소를 구현할 수 있었으며 이를 통해 게임플레이의 재미와 몰입감을 높일 수 있었다.

## 2.4. 시행착오 및 해결방안

### 1. 미로.fbx파일 불러오기 오류

처음에는 OpenGL로 미로를 만드는 것 보다 유니티에서 만들어서 텍스처맵핑까지 완료한 후 불러오는 방식을 택하였다. fbx파일을 불러올 수 있는 `assimp library`를 사용하기 위해서서 `assimp`를 `opengl`로 불러오기 위해 헤더파일 경로와 라이브러리 파일 경로를 속성에서 설정하였다. 그러나 `lib`파일이 `c`드라이브에서 보이지 않고 오류가 반복되어 OpenGL에서 직접 미로를 모델링하는 방식을 선택하였다.

### 2. 바닥 텍스쳐 매핑이 안되는 오류



텍스처 매핑과정 중 바닥 텍스처가 매핑이 계속 안되는 문제가 있었다. 처음에는 조명 설정때문이라고 생각하였기 때문에 위에서 아래로 비추는 바닥전용 조명을 추가하는 등 계속 조명 설정을 조정했지만 계속 텍스처가 매핑되지 않는 상태였다. 이 오류는 실행파일 exe파일경로에 바닥.bmp가 없어서 생긴 것으로 경로를 올바르게 설정하자 바로 해결되었다.

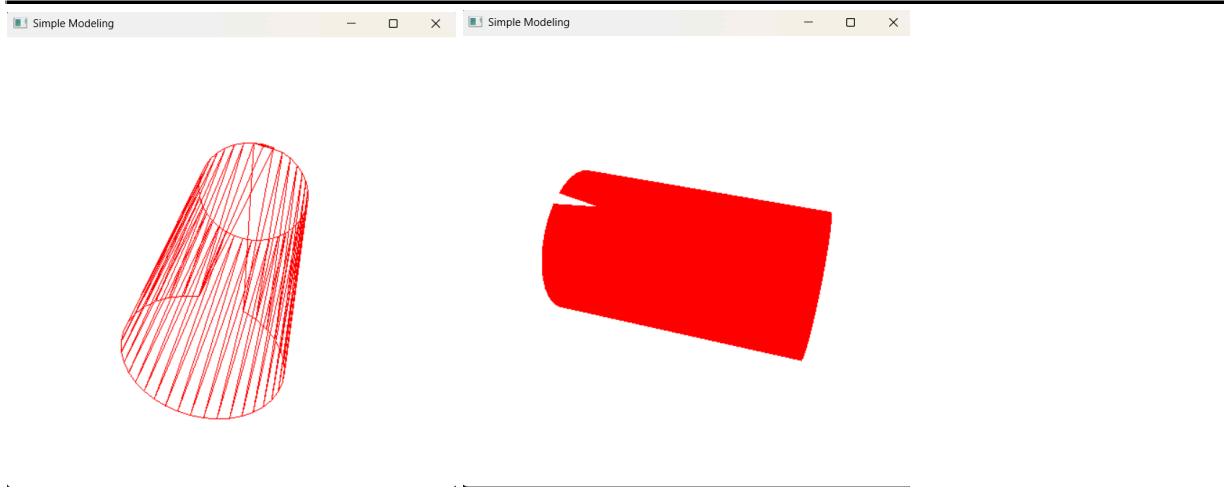
### 3. 오브젝트와의 충돌 체크 오류

초반에 오브젝트와 충돌이 일어날 때 조금 떨어져 있는 상태에서도 오브젝트가 삭제되어 문제가 있었다. 충돌 범위 거리가 문제라고 생각하고 유클리드거리 계산방식으로 테스트하여 플레이어와 오브젝트 위치를 출력해 계산 결과를 확인하였다. 그 과정을 통해  $distance < 0.3f$  설정 시 충돌이 자연스럽게 발생하는 것을 확인하고 적용하였다.

### 4. SOR 프로그램을 통한 오브젝트 생성 시 **polygon** 생성 오류

SOR 프로그램으로 제작 과정에서, 회전 오브젝트가 정상적으로 생성되는지 디버깅하기 위해 평행한 위치로 두 점만 찍어 확인해 보았다. 회전 시 원기둥 모형이 나타나야 한다. 그러나 첫 번째 점과 마지막 점 간의 연결이 이루어지지 않았다.

이는 코드 작성 초기 단계에서, 입력된 점과 회전된 점의 값을 따로 저장하는 코드를 작성했으나, 회전된 점에 초기 점 정보까지 겹쳐 두 번씩 저장된 문제로 인해 발생한 오류였다.



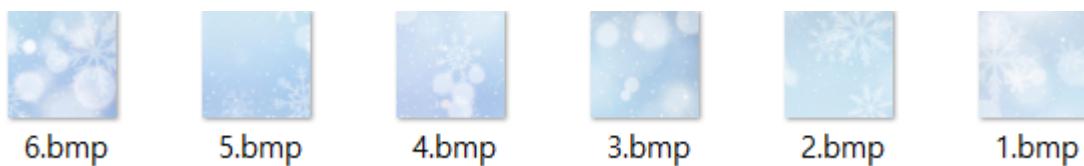
## 5. 미로에 배치된 오브젝트가 보이지 않는 현상

생성된 오브젝트를 미로에 배치시키는 코드를 완성하였으나, 오브젝트와 미로가 모두 보이지 않고 무언가에 덮여있는 듯이 어두운 화면만 계속해서 나타나는 현상이 일어났다. 화면 전체가 오브젝트의 색상으로 미묘하게 가득차 표현되는 것을 확인하여, 오브젝트의 크기가 미로의 크기에 비해 매우 크다는 점을 알 수 있었다. 실제로 SOR 구현 프로젝트와 미로 게임 구현 프로젝트를 둘로 나눠 작업하다 보니, 좌표 크기가 동일하게 설정되지 않은 점으로 발생한 문제라고 판단하였다.

이는 미로 게임 프로젝트 내 오브젝트들을 불어오면서, 각 오브젝트의 `scale` 값 조절을 통해 크기가 매우 작아지도록 수정하였더니 정상적으로 나타날 수 있었다.

## 6. sky 큐브 이미지 텍스처 매핑 오류

눈 이미지 파일을 6면으로 컷팅하여 sky 큐브의 각 면에 6개의 이미지 모두를 텍스처 매핑하는 코드를 작성하여 실행하였으나, 실행 화면이 계속해서 로드되지 않는 현상이 반복되었다. 대표적인 하나의 이미지 파일만 6면에 동일하게 매핑했더니 정상적으로 작동하는 결과를 보였다. 아마 각 이미지마다 파일 크기가 매우 커 로딩되지 않는 것으로 예상된다. 각 이미지의 크기를 `1080*1080`으로 설정했음에도 나타나는 것으로 보아 파일 자체는 문제가 없다고 판단하기 때문이다.



## 7. 외부 실행파일(.exe) 불러오기

본 과제에서는 SOR 실행파일(.exe)에서 점 정보가 저장된 후, 바로 미로 실행파일(.exe)이 불러오도록 설계했다. 이 기작은 크리스마스 미로 게임을 제작한다는 과제 주제에 대해서,

Player가 SOR 화면에서 직접 오브젝트를 제작하는 과정까지 게임의 흥미 요소로서 작용할 수 있도록 고민한 결과였다.

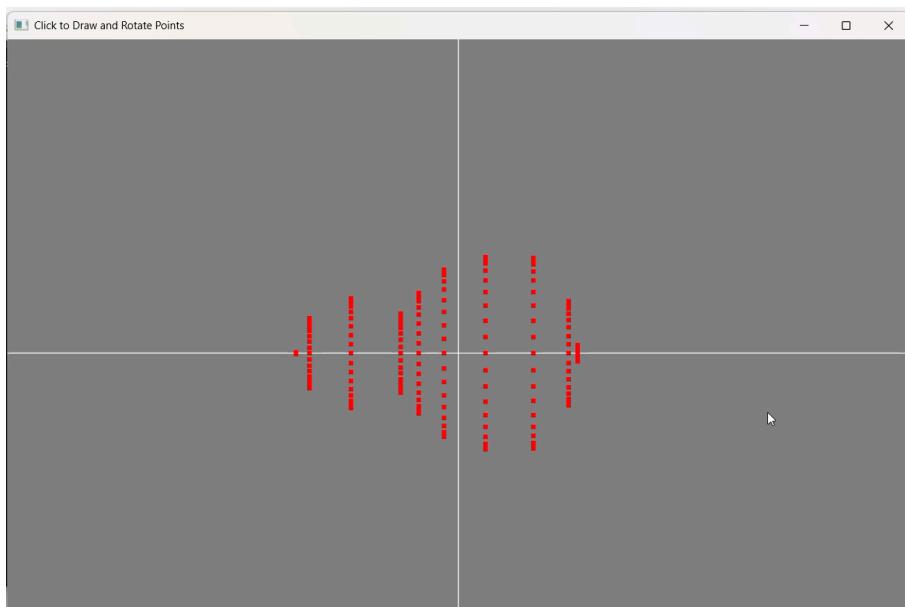
처음에는 두 파일 내 코드를 통합시키고자 하였으나, 두 코드 내 변수들과 구조체 간 서로 통일성이 부족하여 통합시키는 것에는 큰 어려움을 겪어 결국 이러한 방식을 택했다.

두 실행파일이 필요하기에 visual studio 내에서 2개의 solution이 필요했다. 미로 실행 파일에 대한 저장경로 또한 적절하게 설정해야 했기에 생각보다 복잡한 과정을 겪었다.

## 8. 입력한 점을 회전시킨 점에 대한 여러 시행착오

SOR 실행 파일을 끝내면 바로 미로 게임 실행 파일을 로드하도록 설계했다 보니, 회전시킨 점을 SOR 실행 화면에서 표현해내는 시간이 매우 짧았다. 시간을 지연시키는 코드를 계속해서 적용하였으나, 문제가 해결되지 않아 이대로 구현할 수 밖에 없어 아쉬운 점이었다.

또한, 이 회전된 점들에 대해 SOR 실행 화면 상에서 각 점들의 선을 연결하여, 3차원으로 구현하지 못한 점에 대해 아쉽게 생각한다. 3차원 형태를 확인하려면 SOR 화면에서는 확인하지 못하고, 직접 미로 화면까지 실행해야 한다는 점이 생각보다 불편한 설계였다고 생각한다. 이러한 부분이 더 보완할 점이라고 생각한다.



## 3. 결론

본 프로젝트를 통해 OpenGL을 활용하여 크리스마스 테마의 미로 게임을 설계하고 구현할 수 있었다. 기본적인 SOR 화면 출력에서부터 모델링 데이터 저장 및 불러오기, 미로 구현, 오브젝트 배치, 충돌 처리, 사운드 효과 등 다양한 기능을 종합적으로 구현하였다.

---

먼저, SOR 화면 출력과 모델링 데이터의 저장 및 활용 방식을 통해 데이터 관리의 효율성을 높일 수 있었다. 또한 이 과정을 통해 OpenGL을 통한 모델링 방법에 대한 전반적인 이해를 할 수 있게 되어 다음 과정인 미로 구현을 좀 더 수월하게 진행할 수 있게 되었다.

다음으로 미로 구현 과정에서는 미로의 그래픽 표현, 텍스처 맵핑, 카메라 설정 등 OpenGL의 기본 기능을 적절히 활용하여 3D 환경을 구성하였다. 충돌 처리 및 이벤트 구현을 통해서 플레이어와 오브젝트 간의 상호작용을 자연스럽게 표현할 수 있었다. 또한, 크리스마스 테마를 반영한 배경 음악, 효과음, 성공 음악은 게임의 몰입도를 높이고 특색있는 미로 컨셉을 살리는데 많은 도움이 되었다.

프로젝트 진행 과정에서는 텍스처 맵핑 오류, 오브젝트 충돌 범위 문제, FBX 파일 불러오기 실패 등 다양한 시행착오를 겪었으나, 문제를 분석하고 해결 방안을 찾는 과정을 겪으며 OpenGL에 대한 이해도와 기술을 자연스럽게 흡수할 수 있었다. 특히, 텍스처 맵핑 오류는 실행 파일 경로 문제로 인한 단순 실수였지만, 이를 해결하면서 작업 환경의 중요성을 다시 한번 깨닫게 되었다.

결과적으로, 본 과제를 통해 OpenGL을 활용한 3D 모델링과 게임 개발의 전체적인 흐름을 이해하고, 이를 실제로 구현을 이끌어낼 수 있었다. 또한 크리스마스 테마를 중심으로 한 특색있는 접근과 기술적 구현을 조화롭게 결합하여, 시각적·청각적 몰입감을 제공하는 완성도 높은 게임을 제작할 수 있었다. 본 과제를 통해 얻은 지식과 경험은 향후 컴퓨터 그래픽스 및 게임 개발 프로젝트에서도 유용하게 쓰일 지식이 될 것이다.