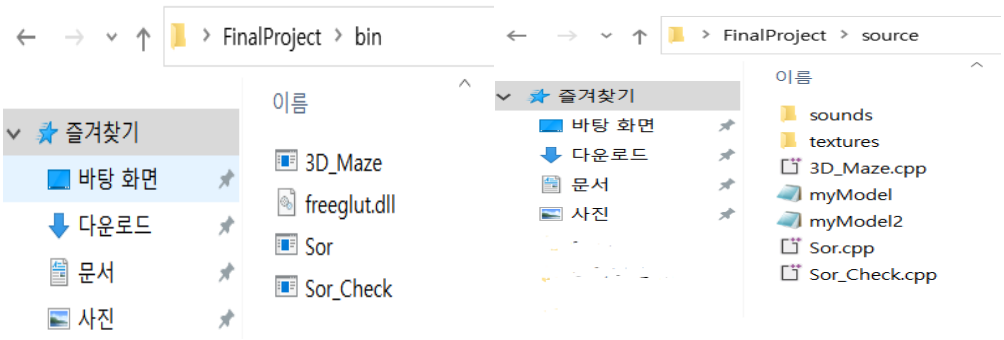
컴퓨터 그래픽스 FINAL PROJECT

20201607 임유민, 20202074 오진우

0. 제출 파일 구조

프로젝트 보고서에 앞서 파일 배포파일 구조에 대한 설명을 하겠습니다. 먼저 bin 파일과 source 파일이 있고, bin 파일은 저희 프로젝트의 exe 파일이 있습니다. 3D_Maze.exe 파일은 미로 탐험 프로그램, Sor은 vertexs 회전 및 모델 저장 프로그램, Sor_Check 는 Sor.exe 파일 내부에서 실생시키는 외부 exe 파일입니다. opengl 라이브러리가 없는 컴퓨터에서도 실행되도록 bin 내부에 freeglut.dll 파일을 추가했습니다.



source 파일안에는 프로젝트의 cpp 파일들이 존재하고, Sor.exe 프로그램으로 미리 생성한 모델링 데이터 2개가 존재합니다. sounds 파일과 textures 파일 안에는 저희 프로젝트 실행파일에서 불러오는 이미지, 음원 파일이 존재합니다.

1. sor 프로그램 구현

1-1. 뷰포트 설정 함수

SOR 모델러를 제작할 때 가장 먼저 해결해야 할 목표는 우선 사용자가 마우스를 입력할 뷰포트를 구성하는 것이었습니다.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(0.0, 0.0, 0.0, 0.0, 0.0, -1.0, 0.0, 1.0, 0.0);

glutInitWindowSize(500, 500);
```

변환의 종류를 명시하기 위해 glMatrixMode 함수의 매개변수를 GL_MODELVIEW로 설정하였습니다. 그 뒤 변환행렬을 초기화시켰습니다. 점을 찍기 위한 x,y 평면을 완전히 뷰포트와 평행하게 설정하기 위해서 gluLookAt 함수를 사용하여 카메라가 0, 0, 0 에서 y축 방향으로 서서 z축을 바라보도록 설정하였습니다. 또한 뷰포트의 크기를 500, 500으로 설정하였습니다.

```
glBegin(GL_LINES);
glVertex3f(500.0, 0.0, 0.0);
glVertex3f(-500.0, 0.0, 0.0);
glVertex3f(0.0, -500.0, 0.0);
glVertex3f(0.0, 500.0, 0.0);
glVertex3f(0.0, 0.0, -500.0);
glVertex3f(0.0, 0.0, 500.0);
glEnd();
```

사용자가 점을 입력할 때 기준이 될 수 있도록 뷰포트에 x축과 y축을 교차하는 십자선을 그렸습니다.

1-2. 점 저장 함수

이제 뷰포트를 만들었으니 점을 입력할 차례입니다. 점을 입력받아 회전시키고 저장하는 것이 SOR모델러의 목표이지만 가장 먼저 점을 입력받기 위해선 우선 점을 저장할 자료구조가 필요했습니다. 그래서 교수님이 강의록에 제시해주신 자료구조를 참고하여 점을 저장할 자료구조를 작성하였습니다.

```
struct Point {

    GLfloat x, y, z;

};
```

3차원 x, y, z 좌표를 저장하고 관리하기 위한 Point 구조체를 만들었습니다. 이 SOR 모델러에선 점들이 몇 백개씩 사용됩니다. 따라서 점들에 대한 point 변수를 for문을 통해 일일히 만들었다가 없앴다 하는 것은 매우 비효율적이라고 생각했습니다. 그래서 저희 조는 point 변수를 담아서 한 꺼번에 관리할 수 있는 백터 컨테이너를 생성하여 점의 좌표 데이터를 효율적으로 다루고자 하였습니다. 사용자가 클릭한 점들을 저장하기 위한 clickedPoints 라는 벡터 컨테이너를 선언하였습니다.

```
vector<Point> clickedPoints;
```

점을 저장하는 자료구조를 생성하였으니 점을 입력받는 함수를 만들어야 했습니다. 점의 좌표를 뷰포트 상에서 마우스 좌클릭을 통해 순서대로 입력받아 clickedpoints 배열에 추가하고자 하였습니다.

```
// mouseClick 함수는 마우스 클릭 이벤트를 처리합니다.
void mouseClick(int button, int state, int x, int y) {
    // 왼쪽 마우스 버튼이 눌렸을 때의 이벤트를 처리합니다.
    if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN) {
```

```
        // 새로운 점(Point) 객체를 생성하고 마우스 클릭 위치를 기준으로 좌표를
        설정합니다.
        Point newPoint;
        newPoint.x = static_cast<GLfloat>(x);            // 마우스 X 좌표
        newPoint.y = static_cast<GLfloat>(500 - y);      // 마우스 Y 좌표
        newPoint.z = 0.0;                                // Z 좌표는 0으로 설정

        // 생성된 점을 clickedPoints 벡터에 추가합니다.
        clickedPoints.push_back(newPoint);
    }

    // 화면 갱신을 요청합니다.
    glutPostRedisplay();
}
```

마우스 좌클릭을 하게되면 제일 먼저 newPoint라는 x, y, z 좌표를 가진 Point 구조체가 생성됩니다. 마우스로 클릭한 x, y 좌표값을 newPoint에 입력받은 뒤에 push_back 함수를 통해 clickedPoints 배열의 제일 마지막에 newPoint x, y 좌표가 추가됩니다. 이렇게 마우스 클릭을 할 때마다 clickedPoints 배열에 사용자가 입력한 좌표값들이 차례대로 추가됩니다.

1-3. 점 회전 각도 저장 함수

점을 클릭하고 저장하는 것은 구현이 되었습니다. 다음 단계로는 사용자가 입력한 각도에 따라 점들을 특정한 축을 기준으로 회전시킬 차례입니다. 저희 조는 우선 360도 회전시킨 점들을 총 몇 번 저장할지를 결정하는 변수를 만들고자 하였습니다. 각도값으로 저장하면 매번 360에서 각도값으로 나눠주어야 했고 각도값을 저장하는 것보다 직관적이고 계산할 때에도 편리하다고 생각하였습니다.

```
int sweepResolutionMod = 6;
```

sweepResolutionMod 변수가 뜻하는 것은 점을 360도 회전시킬 때 점을 몇 번 찍을지 결정하는 값을 의미합니다. 이 코드에서는 6으로 초기화가 되어있는데 즉 360도 회전할때 6번 점을 찍겠다는 것을 뜻합니다.

```
case 'r':
    cout << "360의 약수로 각도를 입력하십시오: ";
    cin >> sweepResolutionMod;
    sweepResolutionMod = 360 / sweepResolutionMod ;
    glutPostRedisplay();
    break;
```

키보드에서 r 키를 누르면 위 코드가 실행됩니다. 콘솔창에 360의 약수로 입력하라는 메시지가 나오면 사용자가 값을 입력합니다. 사용자가 30을 입력하였다고 가정하면 360/30 즉 한바퀴를 도는 동안 점을 총 12번 저장한다는 의미이므로 360을 사용자가 입력받은 값으로 나누어 sweepResolutionMod 변수에 저장합니다.

1-4. 점 회전 및 시각화 함수

점과 각도를 입력 받았으니 이제는 점을 회전시키는 일만 남았습니다. revolutionSurface() 함수의 내부는 크게 이중 for문으로 구성되어 있습니다. 우선 sweepResolutionMod의 값만큼 외부 for문을 반복합니다.

```
for (int i = 0; i <= sweepResolutionMod; i++)
    double theta = i > 0 ? (360 / sweepResolutionMod) * i : 0;
    theta = theta * (M_PI / 180);
```

theta라는 변수는 반복문 안에서 회전 행렬식을 계산할 때 쓰일 각도값를 의미합니다. 일반적으로 OpenGL에서 삼각함수를 사용할 때 각도를 라디안으로 사용합니다. 따라서 각도를 180으로 나누고 π를 곱하여 라디안으로 변환해야 했습니다.

```
for(int i=0; i<arPoints.size(); i++)
{
    xPoint3D newpt;
    newpt.x = arPoints[i].x;
    newpt.y = arPoints[i].y * cos(radian) - arPoint[i].z * sin(radian);
    newpt.z = arPoints[i].y * sin(radian) + arPoint[i].z * cos(radian);
    rotPoints.push_back(newpt);
}
```

```
for (size_t i = 0; i < clickedPoints.size(); i++)
{
    GLfloat tempX = (clickedPoints[i].x - 250) / 250;
    GLfloat tempY = (clickedPoints[i].y - 250) / 250;
    GLfloat tempZ = 0.0;

    GLfloat newX = tempX;
    GLfloat newY = (tempY * cos(theta)) - (tempZ * sin(theta));
    GLfloat newZ = (tempY * sin(theta)) + (tempZ * cos(theta));

    glVertex3f(newX, newY, newZ);
}
```

내부 for문은 교수님이 주신 위 힌트를 참고하여 작성하였습니다. 내부 for문은 clickedPoints.size 즉 사용자가 클릭한 횟수만큼 반복합니다. 클릭한 좌표의 x값과 y값에서 250을 빼고 250으로 나누는 것은 회전 행렬식을 적용하기 위해 정규화 하는 과정입니다. 회전 행렬 변환이 끝난 후에 glVertex3f 함수를 통해 변환된 좌표를 뷰포트에 나타내었습니다.

1-5. 회전된 점 저장 함수

점들을 회전시켰으니 저장을 해야합니다. 앞서 봤던 함수와 비슷한 구성으로 되어있습니다. 파일은 vertex부분과 face 부분으로 저장됩니다. 교수님이 참고하라고 알려주신 SOR 모델링 데이터 저장 함수를 저희 코드에 적용시키려고 하니 호환이 잘 되지 않고 계속하여 오류가 발생하였습니

다. 그래서 교수님의 코드에서 for문을 제외한 부분만 가져다 쓰고 for문과 그 내부는 저희 조가 직접 고민하고 설계하여 코드를 작성해보았습니다. 우선 Vertex 점을 저장하는 부분은 이중 for문으로 구성되어 있습니다. sweepResolutionMod의 원소 개수만큼, 즉 클릭한 점의 개수만큼 외부 for문이 반복됩니다. 그리고 점들이 회전하는 횟수만큼 내부 for문이 반복됩니다. 따라서 사용자 클릭한 점들을 회전시킨 좌표들이 전부 저장될 수 있게 알고리즘을 구현해냈습니다.

```
for (size_t i = 0; i < clickedPoints.size(); i++) {
    for (int j = 0; j <= sweepResolutionMod; j++) {
      }
}
```

for문의 내부를 살펴보면 회전 행렬을 적용하는 코드가 있습니다. 따라서 각도를 라디안으로 변환하여 저장하는 theta 변수가 필요하고 클릭한 점들을 회전 행렬식 계산을 위해 정규화하는 과정이 필요합니다. 또한 정규화를 마치고 회전 행렬 변환 계산을 하는 부분 또한 필요합니다.

```
double theta = (360.0 / sweepResolutionMod) * j;
theta = theta * (M_PI / 180);

GLfloat tempX = (clickedPoints[i].x - 250) / 250;
GLfloat tempY = (clickedPoints[i].y - 250) / 250;
GLfloat tempZ = 0.0;
 GLfloat newX = tempX;
GLfloat newY = (tempY * cos(rotationAngle + theta)) - (tempZ *
 sin(rotationAngle + theta));
GLfloat newZ = (tempZ * cos(rotationAngle + theta)) + (tempY *
 sin(rotationAngle + theta));
```

OpenGL에서 회전행렬식 계산을 위한 정규화 과정을 거쳤기에 좌표가 변환된 채 그대로 저장하면 오류가 발생하게 됩니다. 그러므로 정규화하기 전으로 좌표값을 복원하는 과정이 필요하게 됩니다. 정규화 이전 좌표로 돌아가도록 변환된 좌표값에 250을 곱한 뒤 250을 더해줍니다. 그 후 파일에 저장하면 좌표가 올바르게 저장됩니다.

```
fprintf(fout, "%.1f %.1f %.1f\n", newX * 250 + 250, newY * 250 + 250,
 newZ * 250 + 250);
```

이제 파일을 저장하는 함수에서 삼각형 면을 저장하는 부분을 저희조가 직접 구현해야 합니다. 저희 조가 직접 vertex의 인덱스를 저장할 수 있는 알고리즘을 만들었습니다. 이것 역시 이중 for문으로 구성되어 있습니다. 반복 알고리즘은 외부 for문은 clickedPoints.size 만큼 반복하지 않고 -1을 하여 반복합니다. 면이 생성되면 처음 면을 그리기 시작한 점과 마지막 점이 겹치게 되므로 불필요한 면이 생성됩니다. 그래서 총 클릭한 횟수에서 1을 빼서 반복하도록 코드를 작성했습니다. 내부 for문은 점이 회전한 횟수만큼 반복합니다. 따라서 삼각형 면을 구성하는 점들의 인덱스를 빠짐없이 모두 저장할 수 있게 됩니다.

```
for (size_t i = 0; i < clickedPoints.size() - 1; i++) {
    for (int j = 0; j < sweepResolutionMod; j++) {
```

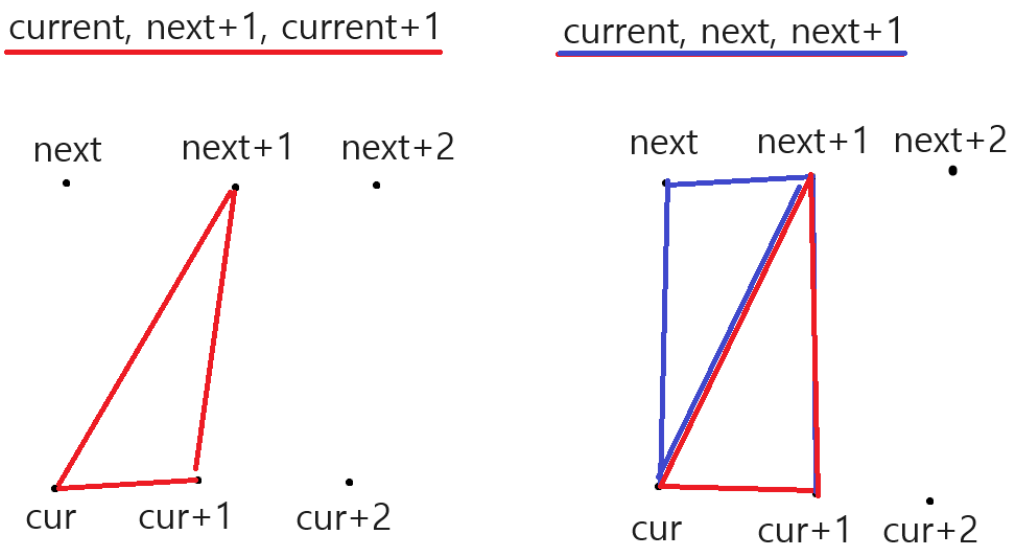
```
        int current = i * (sweepResolutionMod + 1) + j;
        int next = (i + 1) * (sweepResolutionMod + 1) + j;
        fprintf(fout, "%d %d %d\n", current, next + 1, current + 1);
        fprintf(fout, "%d %d %d\n", current, next, next + 1);
    }
}
```

면을 저장하는 것에 있어서 가장 걸림돌이 되었던 것은 어떤 방식으로 점들의 순서를 결정하느냐였습니다. 그 로직을 고민하던 중 저희 조가 생각해 낸 방법은 점들의 층을 나누어서 위아래로 연결시킨다는 방법이었습니다. 점이 y축으로 회전한다고 했을 때 점을 클릭하면 x축으로 평행하게 점들이 저장되게 됩니다. 이렇게 저장된 점의 인덱스를 current 변수에 저장합니다. 그리고 다음 클릭을 하면 current의 위 또는 아래에 점들이 평행하게 생성되고 이 점들의 인덱스를 next에 저장합니다.

```
int current = i * (sweepResolutionMod + 1) + j;
int next = (i + 1) * (sweepResolutionMod + 1) + j;
```

이렇게 점들의 순서를 층을 나누어 표현한 뒤에 current와 next를 연결하여 삼각형을 표현하도록 알고리즘을 직접 만들어 보았습니다. 실제로 점의 인덱스를 파일에 저장하는 코드는 다음과 같습니다.

```
fprintf(fout, "%d %d %d\n", current, next + 1, current + 1);
fprintf(fout, "%d %d %d\n", current, next, next + 1);
```



코드가 점들을 어떻게 저장하는지 이해하기 쉽도록 그림으로 표현해 보았습니다. for문이 한 번 반복될 때마다 위 그림처럼 해당하는 점을 기준으로 면을 생성하는 점의 인덱스들이 저장됩니다. 이 과정이 반복됨으로써 모든 면을 저장할 수 있게 됩니다.

1-6. 회전 축 변경 함수

이 SOR 모델러에서 y축 뿐만 아니라 x, z축에 대해서도 회전할 수 있도록 하는 추가적인 기능을 구현하였습니다. 이를 구현하는 것은 생각보다 간단했습니다. 점들을 뷰포트에 그리거나 저장하는 함수에서 회전 행렬식을 사용하는 부분 앞에 if문을 삽입해 분기점을 만들어서 x,y,z 인지에 따라 그에 해당하는 회전 행렬 변환을 적용하면 되었습니다. 이를 위해서는 x,y,z 어떤 축을 기준으로 회전할지 결정하는 변수를 먼저 만들어야 했습니다.

```
enum Menu {
    SET_ANGLE,
    MENU_MODE_X,
    MENU_MODE_Y,
    MENU_MODE_Z,
    SAVE_MODE,
    MENU_CLEAR_POINTS,
    MENU_EXIT
};

int rotationMode = MENU_MODE_Y;
```

rotationMode 변수는 어떤 축을 기준으로 회전시킬지를 결정하는 변수입니다. 그리고 열거자인 MENU_MODE_Y는 y축을 기준으로 점들을 회전하겠다는 의미를 가집니다. 이 코드에서는 어떤 축으로 회전할지 선택하지 않으면 y축을 기준으로 회전하도록 초기화되어 있습니다. 실제로 분기점을 만들어 점을 그리는 함수에 적용시킨 부분은 다음과 같습니다.

```
if (rotationMode == MENU_MODE_X)
    {
        for (size_t i = 0; i < clickedPoints.size(); i++)
        {
            ...//생략
        }
        glEnd();
    }
    else if(rotationMode == MENU_MODE_Y)
    {
        for (size_t i = 0; i < clickedPoints.size(); i++)
        {
            ...//생략
        }
        glEnd();
    }
    else if(rotationMode == MENU_MODE_Z)
    {
        for (size_t i = 0; i < clickedPoints.size(); i++)
        {
            ...//생략
        }
        glEnd();
    }
```

1-7. 키보드 입력 처리

함수의 점들은 마우스로 입력 받을 수 있지만 각도값을 마우스로 입력받기엔 어려움이 있습니다. 그래서 저희 조는 숫자 데이터인 각도값을 키보드 숫자키를 통해 입력받고자 하였습니다. 또한 데이터를 저장하는 기능과 프로그램을 종료하는 필수적인 기능들을 키보드를 통해 실행할 수 있도록 keyboard 함수를 구현했습니다.

```
// 키보드를 통해 각도를 입력받고 데이터를 저장하는 함수
void keyboard(unsigned char key, int x, int y)
{
    switch (key)
    {
    // s 키를 누르면 현재 점들을 저장함
    case 's':
        SaveModel();
        break;

    // r 키를 누르면 콘솔창에 각도를 입력받음
    case 'r':
        cout << "360의 약수로 각도를 입력하십시오: ";
        cin >> sweepResolutionMod;
        sweepResolutionMod = 360 / sweepResolutionMod ;
        glutPostRedisplay();
        break;

    // ESC 키를 누르면 프로그램 종료
    case 27:
        exit(0);
        break;
    }
}
```

1-8. 팝업 메뉴 함수

추가적으로 SOR 모델러를 사용하는 유저의 편의를 위해서 팝업 메뉴를 만들고 싶었습니다. 팝업 메뉴가 키보드보다 나은 점은 기능에 해당되는 키를 외울 필요가 없다는 장점이 있었기 때문입니다. 기존 glut라이브러리에 존재하는 glutCreateMent 함수를 사용하여 쉽게 이를 구현할 수 있었습니다.

```
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

우선 뷰포트에서 우클릭을 하면 메뉴가 팝업되도록 설정했습니다.

```
glutAddMenuEntry("Set Rotation Angle", SET_ANGLE);
glutAddMenuEntry("Set Rotation Axis to X", MENU_MODE_X);
glutAddMenuEntry("Set Rotation Axis to Y", MENU_MODE_Y);
glutAddMenuEntry("Set Rotation Axis to Z", MENU_MODE_Z);
glutAddMenuEntry("Save the data", SAVE_MODE);
glutAddMenuEntry("Clear Points", MENU_CLEAR_POINTS);
glutAddMenuEntry("Exit the SOR", MENU_EXIT);
```

우클릭을 하면 팝업되는 메뉴들 또한 만들어야 했습니다. 메뉴 항목은 회전시킬 각도를 정하는 메뉴, 어떤 축으로 회전할지를 결정하는 메뉴, 결과를 저장하는 메뉴, 입력한 점들을 초기화시키는 메뉴와 SOR 모델러를 종료하는 메뉴로 구성되어 있습니다. 아래는 createMenu 함수의 전체 코드입니다. switch 문을 사용하여 팝업 메뉴에서 특정한 버튼을 선택함에 따라 그 메뉴에 알맞은 동작이 실행되도록 구현하였습니다.

```
void createMenu() {
    glutCreateMenu([](int value) {
        switch (value) {
        case SET_ANGLE:
            cout << "360의 약수로 각도를 입력하십시오: ";
            cin >> sweepResolutionMod;
            sweepResolutionMod = 360 / sweepResolutionMod;
            glutPostRedisplay();
            break;
        case MENU_MODE_X:
            ClearPoints();
            rotationMode = MENU_MODE_X;
            break;
        case MENU_MODE_Y:
            ClearPoints();
            rotationMode = MENU_MODE_Y;
            break;
        case MENU_MODE_Z:
            ClearPoints();
            rotationMode = MENU_MODE_Z;
            break;
        case SAVE_MODE:
            SaveModel();
            break;
        case MENU_CLEAR_POINTS:
            ClearPoints();
            break;
        case MENU_EXIT:
            exit(0);
            break;
        }
        glutPostRedisplay();
        });
    glutAddMenuEntry("Set Rotation Angle", SET_ANGLE);
    glutAddMenuEntry("Set Rotation Axis to X", MENU_MODE_X);
    glutAddMenuEntry("Set Rotation Axis to Y", MENU_MODE_Y);
    glutAddMenuEntry("Set Rotation Axis to Z", MENU_MODE_Z);
    glutAddMenuEntry("Save the data", SAVE_MODE);
    glutAddMenuEntry("Clear Points", MENU_CLEAR_POINTS);
    glutAddMenuEntry("Exit the SOR", MENU_EXIT);
    glutAttachMenu(GLUT_RIGHT_BUTTON);
}
```

1-9. sor 생성 모델 확인 기능

저희는 저희가 sor 프로그램으로 생성란 모델이 잘 생성 되었는지를 수시로 확인해야할 필요가 있었습니다. 이는 교수님께서 주신 예제파일에서 생성된 모델의 3d 형태를 확인할 수 있는 기능들만 남기고 지워서, 구성하였습니다. 3d 로 확인 하는 기능은 기존 sor 코드 안에 넣지 못했고, 외부 exr 파일로 변경후, 이 exe 파일을 실행시켜서 확인하는 방법을 이용했습니다. 키보드 이 exe 프로그램을 실행시켜, 생성한 모델을 확인하는 방법은 f 키를 누르면 됩니다. (교수님의 예제코드와 거의 동일하여 보고서에는 설명을 생략하고, 외부 exe 파일 실행 함수를 올렸습니다.)

```
// sor 모델링 파일 확인 exe 파일 실행 함수
void runExeFile(const char* exePath) {
    system(exePath);
}
```

1-10. 구현한 기능 정리

1. 점들을 마우스 클릭을 통해 입력받는 기능
2. 점들을 회전할 각도를 입력받는 기능
3. 점들을 특정한 축을 기준으로 회전시키는 기능
4. 회전한 점들 파일에 저장하는 기능
5. 회전시킬 축을 선택할 수 있는 기능
6. 입력한 점들을 초기화 하는 기능
7. 팝업 메뉴 기능

단축키:
R: 회전 각도 수동 입력
S: 모델링 확인 프로그램 실행
ESC: 프로그램 종료

1-11. 시현 기능 사진

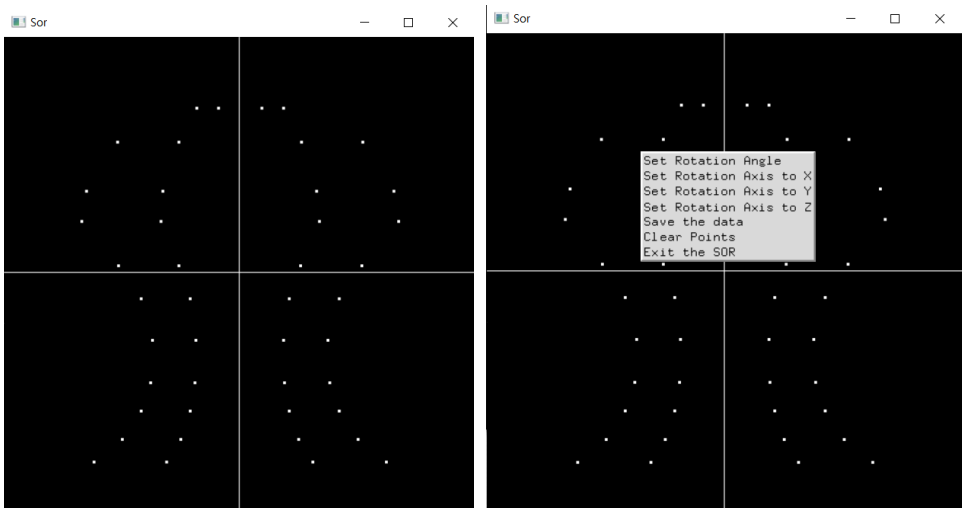


그림 1,2 y축 기준 회전, 메뉴창 화면

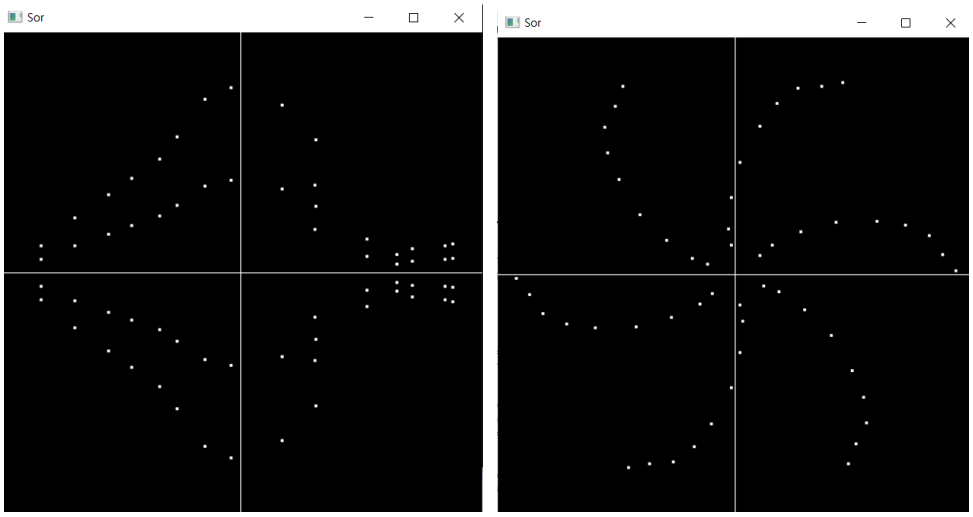


그림 2,3 x, z축 기준 회전

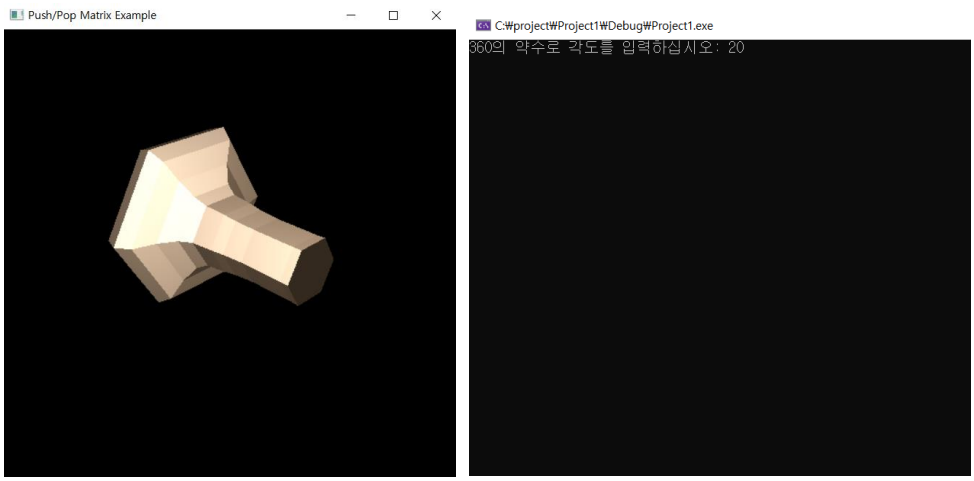


그림 4,5 sor 생성 모델 확인, sor 각도 입력 창

2. 3d mazw 탐험 프로그램 구현

2-1. 카메라 위치 업데이트 함수

3d 공간에 미로를 제작할 때 가장 먼저 해결해야할 목표를 3d 공간에서의 자유로운 카메라의 이동이였습니다.

```
void gluLookAt(double eyeX, double eyeY, double eyeZ,
               double centerX, double centerY, double centerZ,
               double upX, double upY, double upZ);
```

gluLookAt 함수에서 keyeX,Y,Z 매개변수는 카메라의 위치, centerX,Y,Z 는 카메라가 바라보는 점,

그리고 upX,Y,Z 는 카메라의 업벡터를 나타냅니다. 카메라의 위치는 이벤트 처리를 하여 움직이게 하면 그만이였지만, 카메라가 바라보는 방향을 회전해야하는데, 그렇게 된다면 카메라의 바라보는 시점이 계속해서 변화하여 값이 계속해서 변하는 문제가 있었습니다. 이 카메라가 바라보는 시점을 수학적으로 유도해보면

```
float lookAtX = cameraX + 5.0f * sin(+cameraAngleY) * cos(cameraAngleX);
float lookAtY = cameraY + 5.0f * sin(-cameraAngleX);
float lookAtZ = cameraZ - 5.0f * cos(+cameraAngleY) * cos(cameraAngleX);
```

sin(+cameraAngleY) * cos(cameraAngleX): 이 부분은 YZ 평면에서의 회전 효과를 나타냅니다. cameraAngleY는 Y 축 주위의 회전을 나타내고, cameraAngleX는 X 축 주위의 회전을 나타냅니다. 그리고 sin(-cameraAngleX): 이 부분은 X 축 주위의 회전 효과를 나타냅니다. cameraAngleX가 X 축 주위의 회전을 나타내므로, 이 값을 사용하여 X 축 주위의 회전 효과를 계산합니다. 마지막으로 cos(+cameraAngleY) * cos(cameraAngleX): 이 부분은 YZ 평면에서의 회전과 X 축 주위의 회전을 동시에 고려한 효과를 나타냅니다. cos(+cameraAngleY)는 Y 축 주위의 회전에 영향을 주고, cos(cameraAngleX)는 X 축 주위의 회전에 영향을 줍니다. 따라서 이 부분은 Y 축 주위의 회전과 X 축 주위의 회전을 함께 고려한 효과를 YZ 평면에 적용하는 것을 의미합니다.

```
float cameraX = 0.0f;
float cameraY = 1.0f;
float cameraZ = 5.0f;
```

초기의 카메라 위치 저희는 이렇게 선정했습니다. 따라서 이를 위의 유도식에 적용해 최종 카메라 방향을 회전했을 때의 gluLookAt 매개변수들을 구해 카메라의 회전을 업데이트 하는 함수를 아래와 같이 작성할 수있습니다.

```
// updateCamera 함수는 카메라의 뷰 매트릭스를 업데이트하는 함수입니다.
 void updateCamera() {
     // glMatrixMode 함수로 현재 작업할 매트릭스 모드를 설정합니다.
     // GL_MODELVIEW 모드는 카메라 뷰 및 모델 변환을 위한 모드입니다.
     glMatrixMode(GL_MODELVIEW);

     // glLoadIdentity 함수로 현재 매트릭스를 단위 행렬로 초기화합니다.
     glLoadIdentity();

     // 카메라가 바라보는 위치를 계산합니다.
     // 이 위치는 카메라의 현재 위치와 방향(각도)에 기반합니다.
     float lookAtX = cameraX + 5.0f * sin(+cameraAngleY) * cos(cameraAngleX);
     float lookAtY = cameraY + 5.0f * sin(-cameraAngleX);
     float lookAtZ = cameraZ - 5.0f * cos(+cameraAngleY) * cos(cameraAngleX);

     // gluLookAt 함수로 카메라의 위치와 카메라가 바라보는 위치를 설정합니다.
     // 마지막 세 매개변수(0.0f, 1.0f, 0.0f)는 카메라의 '업' 방향을 나타냅니다.
     gluLookAt(cameraX, cameraY, cameraZ, lookAtX, lookAtY, lookAtZ, 0.0f,
1.0f, 0.0f);
 }
```

2-2. 카메라 위치 방향 업데이트 함수

카메라가 움직이고, 위치를 지속적으로 업데이트 하는 함수를 구현해야합니다.

```
float cameraAngleY = 0.0f;
float cameraAngleX = 0.0f;
float cameraSpeed = 0.005f;
float rotateSpeed = 0.05f;
```

3d 공간에서 굳이 카메라의 y축을 조절할 필요는 없습니다. y 축을 변경해야 할때는 사용자가 점프를 하였을 때 필요합니다. 평소에 3d 공간을 탐험을 할때는 카메라의 x,z 좌표만 변경해주면 됩니다. wasd 키 입력을 통해 앞으로, 뒤로, 좌우로 이동할때, cameraSpeed값을 계속해서 더해 카메라의 움직임을 지정합니다. 그렇다면 단순히 앞뒤로 갈때는 z값에 더하거나 빼고, 좌우로 이동할 때는 x값을 더해주거나 빼는 간단한 형태로 구현이 가능합니다. 하지만 1인칭게임에서는 단순히 앞뒤, 좌우로 움직이는게 아니라 카메라가 바라보는 시점기준에서 움직임 구현이 가능해야합니다. 즉 카메라의 회전으로 방향전환이 가능해야합니다. 이를 구현하기 위해서는 현재 카메라가 바라보는 방향의 각도가 필요하며, 이를 이용해 계산하면 다음과 같은 코드를 구현할 수 있습니다.

```
float moveX = 0.0f;
float moveZ = 0.0f;

    if (moveForward) {
        moveX += cameraSpeed * sin(cameraAngleY);
        moveZ -= cameraSpeed * cos(cameraAngleY);
    }
    if (moveBackward) {
        moveX -= cameraSpeed * sin(cameraAngleY);
        moveZ += cameraSpeed * cos(cameraAngleY);
    }
    if (moveLeft) {
        moveX -= cameraSpeed * cos(cameraAngleY);
        moveZ -= cameraSpeed * sin(cameraAngleY);
    }
    if (moveRight) {
        moveX += cameraSpeed * cos(cameraAngleY);
        moveZ += cameraSpeed * sin(cameraAngleY);
    }

    cameraX += moveX;
    cameraZ += moveZ;
```

x,z 값을 지속적으로 변경하여 xy 평면에서 자유롭게 움직이는 코드를 구현했으니 이제 점프를 하여 y 값을 조절해줘야합니다. 점프 이벤트가 실행이 되면 카메라의 y좌표가 지속적으로 더해져서 올라갔다가 일정 높이가 되면 점프 상태를 비활성화 하고 다시 y 좌표를 지속적으로 감소시켜 원래 위치로 돌아오는 알고리즘을 조건문을 통해 카메라의 점프 구현을 하였습니다.

```
if (jumping) {
        if (cameraY < 2.0f) {
```

```
            cameraY += 0.01f;
        }
        else {
            jumping = false;
        }
    }
    else {
        if (cameraY > 1.0f) {
            cameraY -= 0.01f;
        }
        else {
            cameraY = 1.0f;
        }
    }
```

점프 이벤트가 활성화 되고 jumping=true 이면 y 값을 0.01 씩 계속해서 더하다가 y 가 2.0 이 되면 else 문이 실행되어 jumping=false 로 돌아옵니다. 그리고 이제 다음에 y 가 1.0 이 될때까지 y 값을 반대로 -0.01 씩 빼서 원래의 높이로 돌아오게 합니다. 따라서 이둘을 합친 카메라의 지속적인 움직임을 업데이트 해주는 idle 함수는 다음과 같습니다.

```
bool jumping = false;

// idle 함수는 이벤트가 발생하지 않을 때 호출되어, 지속적으로 변화하는 상태를
업데이트합니다.
void idle() {
    // 카메라 이동에 대한 X, Z 축 변화량을 초기화합니다.
    float moveX = 0.0f;
    float moveZ = 0.0f;

    // 카메라 이동 처리
    if (moveForward) {  // 앞으로 이동
        moveX += cameraSpeed * sin(cameraAngleY);
        moveZ -= cameraSpeed * cos(cameraAngleY);
    }
    if (moveBackward) {  // 뒤로 이동
        moveX -= cameraSpeed * sin(cameraAngleY);
        moveZ += cameraSpeed * cos(cameraAngleY);
    }
    if (moveLeft) {  // 왼쪽으로 이동
        moveX -= cameraSpeed * cos(cameraAngleY);
        moveZ -= cameraSpeed * sin(cameraAngleY);
    }
    if (moveRight) {  // 오른쪽으로 이동
        moveX += cameraSpeed * cos(cameraAngleY);
        moveZ += cameraSpeed * sin(cameraAngleY);
    }

    // 점프 처리
    if (jumping) {
        // 카메라의 Y 축 위치가 2.0f 에 도달하면 점프 상태를 중지합니다.
        if (cameraY < 2.0f) {
            cameraY += 0.01f;  // 점프 중에는 Y 축 위치를 증가시킵니다.
        }
        else {
            jumping = false;  // 최대 높이에 도달하면 점프 상태를 중지합니다.
```

```
        }
    }
    else {
        // 중력 효과 적용: 카메라가 공중에 있으면 Y 축 위치를 감소시킵니다.
        if (cameraY > 1.0f) {
            cameraY -= 0.01f;
        }
        else {
            cameraY = 1.0f;  // 땅에 도달하면 Y 축 위치를 고정합니다.
        }
    }

    // 계산된 이동량을 적용하여 카메라 위치를 업데이트합니다.
    cameraX += moveX;
    cameraZ += moveZ;

    // 화면을 다시 그리도록 요청합니다.
    glutPostRedisplay();
}
```

2-3. 키보드 입력 처리-움직임 정의

이제 이런 카메라의 움직임들을 처리할 수 있는 키보드를 입력받는 함수를 구현해야합니다. 수업 실습에서 배웠듯이 이러한 키보드 입력은 freeglut 를 사용해 다음과 같은 형태의 코드를 사용하는 것 입니다.

```
void keyboard(unsigned char key, int x, int y) {
    switch (key) {
        case 'w'
            break;
    }
}
```

w를 눌렀을 때 앞으로, s를 눌렀을때는 좌, d 는 우, s 는 뒤로 가도록 처리해야합니다. 때문에 wasd 키를 누르면 위 idle 함수에서 움직이는 상태를 정의해주는 moveForawrd, moveBackword, moveLeft, moveRight 를 true 로 바꾸어주면 됩니다. 하지만 여기에 저희는 카메라가 움직일 때 발걸음 소리파일이 재생되는 추가 기능을 구현할 것입니다. 그래서 카메라가 움직이는 상태 즉 isWalking 이 true 가 되면 발걸음 소리가 재생이 되는 PlaySoundFunction이 시행되도록 해야합니다.

```
switch (key) {
    case 'w':
        if (!isWalking) {
            isWalking = true;
            PlaySoundFunction(true);
        }
        moveForward = true;
        break;
    case 's':
        if (!isWalking) {
            isWalking = true;
```

```
            PlaySoundFunction(true);
        }
        moveBackward = true;
        break;
    case 'a':
        if (!isWalking) {
            isWalking = true;
            PlaySoundFunction(true);
        }
        moveLeft = true;
        break;
    case 'd':
        if (!isWalking) {
            isWalking = true;
            PlaySoundFunction(true);
        }
        moveRight = true;
        break;
```

2-4. 키보드 입력 처리-카메라 회전

wasd 키 말고 다른 키들을 입력해 다른 기능들이 시행되도록 하겠습니다. 위에서는 카메라가 바라보는 방향을 계산하고 정의하였으며, 카메라가 바라보는 방향을 고려하여 카메라가 움직였을때의 위치를 업데이트 해주는 함수를 작성했습니다. 카메라가 바라보는 각도는 cameraAngleX,Y에 저장이 됩니다.

```
float cameraAngleY = 0.0f;
float cameraAngleX = 0.0f;
```

카메라가 바라보는 현재 각도는 0 으로 지정되어있습니다. 그래서 현재 카메라의 각도를 조절해주는 이벤트를 작성한다면, 3d 공간에서 카메라가 사용자의 원하는 방향으로 자유롭게 움직일 수 있게 됩니다. 이런 각도를 조절해주는 이벤트는 위에 키보드 입력처리를 통해 구현이 가능합니다.

```
float rotateSpeed = 0.05f;

case 'q':
        cameraAngleY -= rotateSpeed;
        break;
    case 'e':
        cameraAngleY += rotateSpeed;
        break;
```

이와 같이 q와 e 입력을 통해 cameraAngle 값을 조절하여 카메라가 좌우 회전을 할 수 있도록 기능을 추가할 수 있습니다.

2-5. 키보드 입력 처리-카메라 속도 조절

이 프로그램을 다른 컴퓨터에서 실행할 때 컴퓨터의 상황에 따라 움직이는 정도가 다르다는 것을 저희 팀에서는 발견했습니다. 제 노트북에서는 속도가 적당했지만, 저희 조원의 컴퓨터에서는 카메라의 움직이는 속도가 너무 느려 문제가 되었습니다. 그래서 키보드 입력을 통해 cameraSpeed 변수를 조절해 카메라 이동속도를 조절할 수 있도록 하였습니다.

```
case '1':
        cameraSpeed += 0.003f;
        break;
    case '2':
        cameraSpeed = std::max(cameraSpeed - 0.003f, 0.0f);
        break;
```

5. 키보드 입력 처리-점프 구현

```
case ' ':
        JumpSoundFunction(true);
        jumping = true;
         break;
```

이제 스페이스를 누르면 jumping 상태가 true 가 되어서 전에 구현한 점프기능이 시행되도록 해야합니다. 저희 조는 추가적으로 점프를 했을 때 점프 효과음이 나는 기능을 위의 코드를 추가하여 추가했습니다. 때문에 스페이스바 입력 처리를 했을 때 JumpSoundFunction 함수가 시행되도록 하였습니다.
따라서 최종 구현한 keyboard 함수는 다음과 같습니다.

```
// keyboard 함수는 키보드의 키가 눌렸을 때 호출되는 콜백 함수입니다.
void keyboard(unsigned char key, int x, int y) {
    switch (key) {
    case 'w':  // 'w' 키: 앞으로 이동
        if (!isWalking) {
            isWalking = true;
            PlaySoundFunction(true);  // 걷는 소리 시작
        }
        moveForward = true;
        break;
    case 's':  // 's' 키: 뒤로 이동
        if (!isWalking) {
            isWalking = true;
            PlaySoundFunction(true);  // 걷는 소리 시작
        }
        moveBackward = true;
        break;
    case 'a':  // 'a' 키: 왼쪽으로 이동
        if (!isWalking) {
            isWalking = true;
            PlaySoundFunction(true);  // 걷는 소리 시작
        }
        moveLeft = true;
```

```
        break;
    case 'd':  // 'd' 키: 오른쪽으로 이동
        if (!isWalking) {
            isWalking = true;
            PlaySoundFunction(true);  // 걷는 소리 시작
        }
        moveRight = true;
        break;
    case 'q':  // 'q' 키: 카메라를 왼쪽으로 회전
        cameraAngleY -= rotateSpeed;
        break;
    case 'e':  // 'e' 키: 카메라를 오른쪽으로 회전
        cameraAngleY += rotateSpeed;
        break;
    case '1':  // '1' 키: 카메라 속도 증가
        cameraSpeed += 0.003f;
        break;
    case '2':  // '2' 키: 카메라 속도 감소
        cameraSpeed = std::max(cameraSpeed - 0.003f, 0.0f);
        break;
    case ' ':  // 스페이스바: 점프
        JumpSoundFunction(true);
        jumping = true;  // 점프 활성화
        break;
    case 27:  // ESC 키: 프로그램 종료
        exit(0);
        break;
    }

    // 화면을 다시 그리도록 요청합니다.
    glutPostRedisplay();
}
```

2-6. 키보드 입력 처리- 사운드 파일 재생 조건 정의

현재까지 설명한 코드에서는 wasd 키를 눌러서 PlaySoundFunction 함수를 시행해 발걸음 소리가 나도록 하고 있습니다. 하지만 여기에 추가로 방향키들이 눌러졌을 때만 사운드가 재생이 되고, 방향키를 떼었을 경우, 즉 카메라의 움직임이 정지되어 있을 때 발걸음 소리를 중지시키는 기능이 필요합니다. 이러한 기능이 없으면 방향키를 누르면 발걸음 소리가 재생이 되지만, 카메라가 중간에 움직임을 멈춰도 발걸음 소리 음원파일의 재생이 끝날 때 까지 소리가 계속되는 문제가 생깁니다. '특정키를 떼었을 때' 를 처리하는 기능은 키보드 입력을 한번 더 정의해주고, 반대로 움직임 상태를 False 로 재정의해주면 특정키보드가 눌렀다 떼었을때의 이벤트 처리를 할 수 있습니다.

```
case 'w':
    if (!isWalking) {
        isWalking = true;
        PlaySoundFunction(true);
    }
    moveForward = true;
    break;
```

즉 이전에 키보드 입력을 받는 함수에서 w 키를 눌렀을 때는 movefoward 가 true 가 되어 카메라가 앞으로 움직입니다. 이 코드가 실행된 후에 w키를 께면(해당 키 이벤트가 발생하면) 아래의 코드가 실행되면서 moveforward 가 비활성화 됩니다.

```
case 'w':
    moveForward = false;
    break;
```

그럼 이제 카메라가 앞으로 움직이는지 안움직이는지, 뒤로 움직이는지 안움직이는지, 좌우로 움직이는지 안움직이는지를 정의해주는 기능을 구현했습니다. 여기서 카메라가 정지한 상태인 isWalking = false 인 상태는 각 4방향에서의 움직임을 정의하는 moveforward, movebackward, moveleft, moveright 가 전부 false 일때 입니다. 따라서 아래와 같은 조건문으로 카메라가 멈춰있을때를 조건문으로 판별하고 멈춰있을 때 발걸음 소리를 나게 해주는 playsoundfunction 함수가 시행되지 않게, 즉 소리가 멈추게 할 수 있습니다.

```
if (!moveForward && !moveBackward && !moveLeft && !moveRight) {
        isWalking = false;
        PlaySoundFunction(false);
    }
```

따라서 키보드 입력이 떼어져서 카메라가 정지했을 때, 발걸음 소리가 안나게 하는 keyboardup 함수는 다음과 같습니다.

```
// keyboardUp 함수는 키보드의 키가 떼어졌을 때 호출되는 콜백 함수입니다.
void keyboardUp(unsigned char key, int x, int y) {
    switch (key) {
    case 'w':  // 'w' 키: 앞으로 이동 중지
        moveForward = false;
        break;
    case 's':  // 's' 키: 뒤로 이동 중지
        moveBackward = false;
        break;
    case 'a':  // 'a' 키: 왼쪽으로 이동 중지
        moveLeft = false;
        break;
    case 'd':  // 'd' 키: 오른쪽으로 이동 중지
        moveRight = false;
        break;
    }

    // 이동 중이지 않다면 걷는 소리 중지
    if (!moveForward && !moveBackward && !moveLeft && !moveRight) {
        isWalking = false;
        PlaySoundFunction(false);
    }

    // 화면을 다시 그리도록 요청합니다.
    glutPostRedisplay();
}
```

2-7. 마우스 움직임을 통화 화면 회전

저희조의 초기 프로젝트 코드는 카메라의 화면회전을 q,e 키입력을 통해 시행했습니다. 저희는 실제 게임처럼 마우스의 움직임을 이벤트로 처리하여 마우스의 움직임으로 화면의 전환이 가능하도록 기능을 추가했습니다. 이 기능을 구현할 때 필요로 하는건 마우스가 얼만큼 어느방향으로 움직였는지를 계산할 필요가 있었습니다. 그렇기 때문에 마우스의 초기 위치값을 알고, 그 초기 위차값으로부터 얼마나 떨어졌는 지를 계산하면 이를 구현할 수 있습니다. 그래서 이 마우스의 초기위치값을 어떻게 정할까 고민하다 나온 결론은 마우스 클릭으로 이를 해결하는 것이였습니다. 즉 마우스를 클릭하고, 클릭한 상태로 마우스를 움직이면 클리한 초기 마우스 위치값을 기준으로 카메라의 회전 정도를 계산하는 것입니다. 그래서 if(button==GLUT_LEFT_BUTTON) 조건문을 활용하여 마우스를 클릭했을 때의 마우스 좌표를 lastMouseX, lastMouseY 에 저장하는 함수를 작성했습니다.

```
void mouse(int button, int state, int x, int y) {
    if (button == GLUT_LEFT_BUTTON) {
        if (state == GLUT_DOWN) {
            lastMouseX = x;
            lastMouseY = y;
        }
        else if (state == GLUT_UP) {
            lastMouseX = -1;
            lastMouseY = -1;
        }
    }
}
```

여기서 else if 문에서 마우스 초기 좌표값을 -1 로 지정해주는 부분은 마우스를 떼었을 때 초기 마우스 위치를 초기화 하는 기능입니다. 이제 마우스 초기값을 저장해주었으니, 이를 바탕으로 마우스가 얼마나 움직였는지를 계산해야합니다. 이러한 계산은 마우스가 계속해서 눌러졌을 때 진행이 되어야합니다. 전에 else if 문에서 마우스 클릭이 떼어지면 초기 마우스 좌표값이 -1,-1 로 초기화가 되었습니다. 즉 반대로 마우스 초기값이 -1, -1 가 아닌 상태일때의 if 문을 구현하면 마우스가 클릭한뒤 떼지 않고 움직였을때를 판별할 수 있습니다.

```
if (lastMouseX != -1 && lastMouseY != -1) {

}
```

그럼 이제 현재 마우스 위치와 이전 마우스 위치의 차를 통해 마우스 움직임을 계산합니다. 그리고 이 움직인 정도를 통해 카메라 각도를 계산합니다. 카메라 회전은 y 축 기준으로 좌우를 둘러보는 것과 x축 기준으로 위 아래로 둘러보는 회전 두가지를 통해 자유로운 화면전환이 가능합니다. 즉 마우스 움직임정도를 통해 cameraAngleX,Y 값을 변경해주면 됩니다.

```
float deltaX = x - lastMouseX;
        float deltaY = y - lastMouseY;
        cameraAngleY += (deltaX) * 0.001f;
        cameraAngleX += (deltaY) * 0.001f;
```

그리고 실제 사람은 위아래로 둘러볼 때 목이 이상 각도로 꺾이지 않기 때문에 범위 제한을 통해 이러한 내용도 구현했습니다. 그리고 마지막에 lastMouseX,Y 값을 x,y 로 저장해주어 다음 마우스 이동 이벤트에서 사용이 가능하도록 해줍니다. 따라서 저희가 구현한 mouseMotion 함수는 다음과 같습니다.

```
// mouseMotion 함수는 마우스가 움직일 때 호출되는 콜백 함수입니다.
void mouseMotion(int x, int y) {
    // 이전 마우스 위치가 설정되었는지 확인합니다.
    if (lastMouseX != -1 && lastMouseY != -1) {
        // 마우스의 이동량을 계산합니다.
        float deltaX = x - lastMouseX;
        float deltaY = y - lastMouseY;

        // 마우스의 이동량에 따라 카메라 각도를 조절합니다.
        cameraAngleY += (deltaX) * 0.001f;
        cameraAngleX += (deltaY) * 0.001f;

        // 카메라의 상하 각도가 너무 크지 않도록 제한합니다.
        if (cameraAngleX > 1.5f) cameraAngleX = 1.5f;
        if (cameraAngleX < -1.5f) cameraAngleX = -1.5f;

        // 화면을 다시 그리도록 요청합니다.
        glutPostRedisplay();
    }

    // 현재 마우스 위치를 저장합니다.
    lastMouseX = x;
    lastMouseY = y;
}
```

2-8. 사운드 재생 함수

이제 sound 를 불러오는 함수에 대해 설명하겠습니다. 위에서 점프를 할때나 카메라가 움직일 때 효과음이 나도록 JumpsoundFunction, PlaysoundFunction 을 시행시켰습니다.

```
// PlaySoundFunction 함수는 걷는 소리를 재생하거나 중단합니다.
void PlaySoundFunction(bool start) {
    // 걷는 소리 파일의 경로입니다.
    LPCWSTR soundFilePath = L"../source/sounds/walking.wav";

    if (start) {
        // start 가 true 일 경우, 걷는 소리를 비동기적으로 재생하며, 이를 반복합니다.
        // SND_LOOP 는 소리를 반복 재생하도록 설정합니다.
        sndPlaySound(soundFilePath, SND_ASYNC | SND_FILENAME | SND_LOOP);
    }
    else {
        // start 가 false 일 경우, 현재 재생중인 소리를 중단합니다.
        sndPlaySound(NULL, SND_ASYNC);
    }
}
```

(음원을 불러오는 함수들은 구조가 다 똑같기에, 대표적으로 걷는소리 음원파일을 불러오는 PlaysoundFunction 구조만 첨부하였습니다.)

저희 팀은 window api 를 사용하여 소리를 재생시키는 방법을 선택했습니다. 간단한 방법으로 소리 음원을 불러올 수 있는 장점을 가지고 있었지만, wav 음원 파일만을 가져온다는 단점이 있습니다. 위 함수와 마찬가지로 JumpSoundFunction 도 음원 경로만 다를 뿐 구조는 거의 동일합니다. 그리고 저희 조는 걷는 소리와 점프하는 소리 말고도 배경 음악을 추가하기로 했습니다. 그래서 총 사운드를 불러오는 함수는 PlaysSoundFunction, JumpSoundFunction, BackgroundSoundFunction 3개를 만들었습니다. 그리고 BackgroundSoundFunction 함수는 타 음악재생 함수와 달리 배경음악이기에 Main 함수에 포함하여 시행하였습니다. window api 이용하는 방법은 여러 개의 음원이 동시에 재생된다는 한계점이 존재했습니다. 하지만 구현하는 방법인 간단하여 저희 조는 외부 라이브러리를 따로 다운 받지 않고, 이 방법을 계속 사용하기로 하였습니다.

이제 미로의 벽과 바닥을 배치하는 기능을 구현해야합니다. 벽을 만들어 배치하려면 직육면체의 3d 객체를 배치해야합니다. 이 방법은 컴퓨터 그래픽스 수업에서 정육면체 3d 객체를 생성하고 배치하는 코드를 조금 수정하여 그대로 적용했습니다.

2-9. 텍스쳐 매핑-벽

```
GLfloat MyVertices[8][3] = {
    {-1.5f, -1.5f, 0.5f},
    {1.5f, -1.5f, 0.5f},
    {1.5f, 1.5f, 0.5f},
    {-1.5f, 1.5f, 0.5f},
    {-1.5f, -1.5f, -0.5f},
    {1.5f, -1.5f, -0.5f},
    {1.5f, 1.5f, -0.5f},
    {-1.5f, 1.5f, -0.5f}
};

GLfloat MyColors[8][3] = {
    { 1.0, 1.0, 1.0 },
    { 1.0, 1.0, 1.0 },
    { 1.0, 1.0, 1.0 },
    { 1.0, 1.0, 1.0 },
    { 1.0, 1.0, 1.0 },
    { 1.0, 1.0, 1.0 },
    { 1.0, 1.0, 1.0 },
    { 1.0, 1.0, 1.0 }
};

GLfloat MyTextureCoordinates[4][2] = {
    { 0.0, 0.0 },
    { 0.0, 1.0 },
    { 1.0, 1.0 },
    { 1.0, 0.0 }
};

GLubyte MyVertexList[24] = {
    0, 3, 2, 1,
```

```
    2, 3, 7, 6,
    0, 4, 7, 3,
    1, 2, 6, 5,
    4, 5, 6, 7,
    0, 1, 5, 4
};
```

실습했던 예제와 동일하게 직육면체의 점, 색상, 면 정보를 저장합니다. 예제와 달리 직육면체의 3dd 객체를 생성해야해서 점의 위치 정보를 수정해줍니다. 생성할 직육면체는가로, 높이 3, 세로가 1 인 형태입니다. 이제 이 벽을 디스플레이리스트로 생성하고, glpush, glpop 으로 공간에 생성하는 코드를 구현해야합니다. 하지만 그 이전에 저희 조가 이 벽을 나타내는 직육면체에 어떻게 텍스쳐¸Å핑을 했는지 설명드리겠습니다.

텍스쳐 매핑을 하려면, 이미지 파일을 먼저 불러와야합니다.

```
FILE* file = fopen(filename, "rb");
    if (!file) {
        printf("이미지 불러오기 오류\n");
        exit(1);
    }
```

먼저 이미지 파일을 불러옵니다. 파일이 안불러와졌을 때 오류를 알려려는 부분은 코드를 작성하면서 확인이 필요해 삽입한 코드라 설명은 생략하겠습니다.

```
unsigned char header[54];
    fread(header, sizeof(unsigned char), 54, file);

    width = *(int*)&header[18];
    height = *(int*)&header[22];
    int imageSize = width * height * 3;
    *imagePtr = (GLubyte*)malloc(imageSize);
    fread(*imagePtr, sizeof(GLubyte), imageSize, file);
```

저희는 따로 외부 라이브러리를 사용하지 않고, bmp 파일에서 이미지를 하나씩 직접 불러오는 외부 예제 코드를 참고하여 사용하였습니다. 먼저 bmp 파일의 헤더는 54바이트입니다. 이를 배열로 읽어옵니다. 그리고 헤더의 18,22 번째 위치에는 이미지의 가로와 세로 크키 정보가 담겨있기에 이를 불러옵니다. 그리고 직접 이미지의 가로, 세로, 컬러채널 수 를 곱하여 필요한 메모리 크기를 계산하고 해당 크기만큼 동적 메모리를 할당합니다. 마지막으로 할당한 메모리에 bmp 파일의 픽셀 데이터를 읽어옵니다. 최종적으로 저희가 사용한 bmp 이미지 파일을 불러오는 함수는 아래와 같습니다.

```
// LoadBMP 함수는 BMP 형식의 이미지 파일을 읽어와서 OpenGL 에서 사용할 수 있도록
메모리에 로드합니다.
void LoadBMP(const char* filename, GLubyte** imagePtr) {
    // 파일을 이진 읽기 모드로 엽니다.
    FILE* file = fopen(filename, "rb");

    // BMP 파일의 헤더를 저장할 배열입니다. BMP 헤더는 54 바이트로 구성됩니다.
    unsigned char header[54];
    // 파일에서 54 바이트를 읽어와 헤더 배열에 저장합니다.
    fread(header, sizeof(unsigned char), 54, file);

    // 헤더에서 이미지의 너비와 높이를 추출합니다.
```

```
    // 너비는 헤더의 18 번째 바이트부터 시작하며, 높이는 22 번째 바이트부터
시작합니다.
    width = *(int*)&header[18];
    height = *(int*)&header[22];

    // 이미지의 총 바이트 크기를 계산합니다. BMP 파일은 픽셀 당 3 바이트(RGB)를
사용합니다.
    int imageSize = width * height * 3;

    // 메모리에 이미지 데이터를 저장할 공간을 할당합니다.
    *imagePtr = (GLubyte*)malloc(imageSize);

    // 파일에서 이미지 데이터를 읽어와 할당된 메모리에 저장합니다.
    fread(*imagePtr, sizeof(GLubyte), imageSize, file);

    // 파일을 닫습니다.
    fclose(file);
}
```

이제 사용할 여러 텍스쳐를 생성하고 이미지파일로부터 텍스쳐를 로드하는 역할을 하는 코드를 작성해야합니다.

```
glGenTextures(8, textureIDs);

    const char* filenames[9] = {
        "../source/textures/dungeon_wall3.bmp",  // 앞
        "../source/textures/dungeon_wall3.bmp",  // 상하
        "../source/textures/dungeon_wall1.bmp",  // 왼옆(회전)
        "../source/textures/dungeon_wall2.bmp",  // 오옆(회전 x)
        "../source/textures/dungeon_wall3.bmp",  // 뒤
        "../source/textures/dungeon_wall3.bmp",  // 상하
        "../source/textures/dungeon_floor.bmp",    // 바닥
        "../source/textures/dungeon_floor.bmp",    // 천장
        "../source/textures/title.bmp"
    };
```

사용할 8개의 텍스쳐 아이디를 지정해줍니다.그리고 사용할 이미지 파일의 경로들을 배열에 저장을 합니다. 사용할 이미지는 벽의 상하, 그리고 옆면들에 넣어줄 이미지 6개와 바닥과 천장, 기타 등등에 쓰일 이미지 9개입니다.

```
for (int i = 0; i < 9; ++i) {
        glBindTexture(GL_TEXTURE_2D, textureIDs[i]);
        LoadBMP(filenames[i], &images[i]);

        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_BGR_EXT,
         GL_UNSIGNED_BYTE, images[i]);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    }
```

이후 반복문으로 이 저장된 이미지 파일의 경로를 전에 사용했던 BmpLoad 함수를 통해 하나씩 불러와 텍스쳐를 형성합니다. 그리고 이 불러온 이미지 테이터를 image[i] 에 저장합니다. 그

이후의 코드는 텍스쳐 데이터와 필터링을 설정해주는 코드입니다. GL_LINEAR를 사용하여 선형보간 방법을 적용했습니다.

이제 텍스쳐를 입일 이미지 파일을 불러왔으니 디스플레이 리스트를 생성하여 벽을 렌더링 하는 함수를 구현해야합니다. 디스플레이리스트를 생성해 사용할것이기에 새로운 디스플레이 리스트를 생성합니다. 그리고 이후에 클라이언트 상태 및 텍스처 사용 설정을 해줍니다. 텍스처를 사용할 것이기에 GL_TEXTURE_2D 를 활성화했습니다.

```
WallListID = glGenLists(1);
glNewList(WallListID, GL_COMPILE);
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
glEnable(GL_TEXTURE_2D);
```

이제 물체의 변에 텍스쳐를 바인딩 해야합니다. 그전에 이미지를 어떤 방향으로 그릴지를 정해줘야합니다. 벽의 바깥면에 텍스쳐 바인딩이 되어야하기 때문에 glFrontFace로 정점순서를 시계방향으로 설정해주었습니다. 직육면체에 텍스쳐를 바인딩 해야하기 때문에 바인딩 과정을 반복문을 통해 6번 진행하도록 했습니다. 한 면은 사각형, 즉 4개의 정점으로 이루어져있기에 이 또한 반복문으로 각 면의 정점들을 순회하면 텍스쳐 좌표와 정점을 설정합니다. glTexCoord2fv(MyTextureCoordinates[j]); <- 는 현재 정점에 대한 텍스쳐 좌표를 설정하는 코드이며 glVertex3fv(MyVertices[MyVertexList[i * 4 + j]]);<- 이 코드는 현재 정점의 좌표를 설정하는 코드입니다. 따라서 각 정점을 반복문으로 직접 순회하면서 텍스쳐 정점을 설정하고, 불로온 텍스쳐 파일을 바인딩합니다.

```
glFrontFace(GL_CW);

    for (GLint i = 0; i < 6; i++) {
        glBindTexture(GL_TEXTURE_2D, textureIDs[i]);
        glBegin(GL_QUADS);
        glColor3fv(MyColors[i]);  // Set the color for the current face
        for (GLint j = 0; j < 4; j++) {
            glTexCoord2fv(MyTextureCoordinates[j]);
            glVertex3fv(MyVertices[MyVertexList[i * 4 + j]]);
        }
        glEnd();
    }
```

객체에 이미지를 입히는 과정은 교수님께서 텍스쳐매핑 강의가 올라오기 전에 외부 인터넷 코드를 구글링하여 구현한 방법입니다. 이러한 방법은 bmp 파일 데이터에 직접 접근해 이미지를 직접 불러오고, 원하는 객체의 각 면과 점들을 직접 순회하면서 텍스쳐를 바인딩하는 비효율 적인 방법을 가지고 있습니다. 하지만 이 방법을 전부터 사용했던 지라 이 코드 구성을 그대로 사용하여 텍스쳐매핑을 진행하였습니다. 최종적으로 구현한 텍스쳐매핑 함수는 아래와 같습니다.

```
// MakeTextures 함수는 여러 텍스처를 생성하고 설정하는 함수입니다.
void MakeTextures() {
    // glGenTextures 함수를 사용하여 텍스처 ID 를 생성합니다.
```

```
    // 여기서는 8 개의 텍스처 ID 를 생성합니다.
    glGenTextures(8, textureIDs);

    // 텍스처 파일의 경로를 담고 있는 문자열 배열입니다.
    const char* filenames[9] = {
        "../source/textures/dungeon_wall3.bmp", //벽 이미지 1
        "../source/textures/dungeon_wall3.bmp", //벽 이미지 1
        "../source/textures/dungeon_wall2.bmp", //벽 이미지 1
        "../source/textures/dungeon_wall2.bmp", //벽 이미지 1
        "../source/textures/dungeon_wall3.bmp", //벽 이미지 1
        "../source/textures/dungeon_wall3.bmp", //벽 이미지 1
        "../source/textures/dungeon_floor.bmp", //바닥 이미지 1
        "../source/textures/dungeon_floor.bmp", //천장 이미지 1
        "../source/textures/title.bmp"
    };

    // 각 텍스처에 대해 반복합니다.
    for (int i = 0; i < 9; ++i) {
        // glBindTexture 함수를 사용하여 현재 작업할 텍스처를 바인딩합니다.
        glBindTexture(GL_TEXTURE_2D, textureIDs[i]);

        // LoadBMP 함수를 사용하여 BMP 파일로부터 텍스처 이미지를 로드합니다.
        LoadBMP(filenames[i], &images[i]);

        // glTexImage2D 함수를 사용하여 로드된 이미지 데이터를 OpenGL 텍스처로
        설정합니다.
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_BGR_EXT,
GL_UNSIGNED_BYTE, images[i]);

        // 텍스처 필터링 옵션을 설정합니다. 여기서는 선형 필터링을 사용합니다.
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    }
}
```

2-10. 텍스쳐 매핑-바닥, 천장

이제 벽에 이미지를 입혔으니 바닥과 천장에 이미지를 입혀야 합니다. 코드의 구성은 거의 동일합니다. 하지만 차이점은 먼저 벽에 이미지를 입히는 방법은 미리 전에 지정해주었던 vartices 배열에서 정점 정보를 가져와 순회하는 방식으로 진행되었습니다. 하지만 현재 바닥과 천장은 따로 정점정보를 지정해주지 않았기 때문에 아래의 코드처럼 직접 함수 내에서 정점 정보를 넣어 사각형을 생성하고, 이 점들을 GL_TEXTURE_2D 를 통해 입혀주었습니다.

```
    glEnable(GL_TEXTURE_2D);
    glColor3f(1.0f, 1.0f, 1.0f);
    glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 0.0f);
    glVertex3f(-3.5f, 0.0f, -3.5f);
    glTexCoord2f(0.0f, 1.0f);
    glVertex3f(-3.5f, 0.0f, 3.5f);
    glTexCoord2f(1.0f, 1.0f);
    glVertex3f(3.5f, 0.0f, 3.5f);
    glTexCoord2f(1.0f, 0.0f);
```

```
    glVertex3f(3.5f, 0.0f, -3.5f);
    glEnd();
    glDisable(GL_TEXTURE_2D);
```

2-11. 모델링 파일(.dat) 불러오기&생성

이제 sor 에서 생성한 dat 파일을 불러와야합니다. sor 파일에서 저장한 dat 파일을 불러와 점과 면의 정보를 저장하는 방법은 아래 코드를 보면 알 수 있듯이 수업 예제에서 시행했던 코드를 그대로 가져와서 사용했습니다. (dat 파일을 불러오는 코드는 실습코드를 그대로 사용했으므로 설명은 생략하겠습니다.)

```
// ReadModel 함수는 파일에서 3D 모델 데이터를 읽어오는 함수입니다.
void ReadModel()
{
    // 모델 파일의 경로를 문자열로 정의합니다.
    std::string fname = "..\\source\\myModel.dat";

    // 파일 포인터를 선언합니다. 이것은 파일을 읽기 위한 핸들 역할을 합니다.
    FILE* f1;
    // 임시 문자열 버퍼를 선언합니다. 파일로부터 읽은 문자열을 저장하는 데
사용됩니다.
    char s[81];
    // 반복문에서 사용할 인덱스 변수를 선언합니다.
    int i;

    // 이미 할당된 메모리가 있다면 해제합니다. 이는 이전에 읽은 데이터를 클리어하는
용도입니다.
    if (mpoint != NULL) delete[] mpoint;
    if (mface != NULL) delete[] mface;

    // 파일을 열고, 파일이 열리지 않았다면 오류 메시지를 출력하고 프로그램을
종료합니다.
    if ((f1 = fopen(fname.c_str(), "rt")) == NULL) {
        printf("No file\n");
        exit(0);
    }

    // 파일에서 문자열을 두 번 읽어와서 s 변수에 저장합니다. 이 부분은 파일 형식에
따라 다를 수 있습니다.
    fscanf(f1, "%s", s); // 첫 번째 문자열 읽기
    fscanf(f1, "%s", s); // 두 번째 문자열 읽기

    // 파일에서 정점의 개수를 읽어와 pnum 변수에 저장합니다.
    fscanf(f1, "%d", &pnum);

    // 정점 데이터를 저장할 배열을 동적으로 할당합니다.
    mpoint = new Point[pnum];
    // 모든 정점에 대해 반복하면서 x, y, z 좌표를 파일에서 읽어옵니다.
    for (i = 0; i < pnum; i++) {
        fscanf(f1, "%f", &mpoint[i].x);
        fscanf(f1, "%f", &mpoint[i].y);
```

```
        fscanf(f1, "%f", &mpoint[i].z);
    }

    // 파일에서 면의 개수를 읽어와 fnum 변수에 저장합니다.
    fscanf(f1, "%d", &fnum);

    // 면 데이터를 저장할 배열을 동적으로 할당합니다.
    mface = new Face[fnum];
    // 모든 면에 대해 반복하면서 각 면을 구성하는 세 개의 정점 인덱스를 파일에서
읽어옵니다.
    for (i = 0; i < fnum; i++) {
        fscanf(f1, "%d", &mface[i].ip[0]);
        fscanf(f1, "%d", &mface[i].ip[1]);
        fscanf(f1, "%d", &mface[i].ip[2]);
    }

    // 파일을 닫습니다.
    fclose(f1);
}
```

이제 불러온 모델을 특정 위치와 크기로 그리는 함수에 대해 설명하겠습니다.

```
    glTranslatef(2.0f, -0.2f, 3.2f);
    glScalef(0.003f, 0.003f, 0.003f);
```

모델을 현재 위치에서 X 축으로 2.0 단위, Y 축으로 -0.2 단위, Z 축으로 3.2 단위 이동시키고, 그 다음에 X, Y, Z 각 축으로 각각 0.003 배만큼 크기를 조절합니다.

```
    glColor3f(0.7, 0.2, 0.3);
    glShadeModel(GL_SMOOTH);
    glBegin(GL_TRIANGLES);
```

모델의 섹상을 glColor 함수를 이용하여 모델의 색상을 지정해줍니다. 그리고 GL_TRIANGLES를 통해 모델을 그립니다, 삼각형 으로 모델을 그리는 이유는 dat 모델을 sor 을 저장할 때 삼각형 단위로 점들을 지정해 면을 지정했기 때문입니다.

```
for (int i = 0; i < fnum; i++) {
    glVertex3f(mpoint[mface[i].ip[0]].x, mpoint[mface[i].ip[0]].y,
        mpoint[mface[i].ip[0]].z);
    glVertex3f(mpoint[mface[i].ip[1]].x, mpoint[mface[i].ip[1]].y,
        mpoint[mface[i].ip[1]].z);
    glVertex3f(mpoint[mface[i].ip[2]].x, mpoint[mface[i].ip[2]].y,
        mpoint[mface[i].ip[2]].z);
}
glEnd();
```

이제 ReadModel 함수를 통해 dat 파일에서 불러와 모델에 저장한 점과 면의 정보를 통해 모델을 그립니다. 반복문 안에 각 점을 나타내는 3개의 함수는 한 폴리곤을 구성하는 3개의 정점을 의미합니다. 때문에 mface 에 저장된 3개의 정점 즉 mface[0], mface[1], mface[2] 를 통해 저장된 모델의 정점 정보에 접근하여 모델을 그립니다. 불러온 3d 모델을 화면에 그리는 함수는 아래와 같습니다.

```
// DrawModelAtOffset 함수는 저장된 3D 모델 데이터를 화면에 그리는 함수입니다.
```

```
void DrawModelAtOffset(void)
{
    // 모델을 이동시키는 함수입니다. 여기서는 (2.0f, -0.2f, 3.2f)만큼 이동합니다.
    // 이는 X, Y, Z 축을 따라 모델을 이동시킵니다.
    glTranslatef(2.0f, -0.2f, 3.2f);

    // 모델의 크기를 조정하는 함수입니다. 여기서는 각 축에 대해 0.003 배로
축소합니다.
    glScalef(0.003f, 0.003f, 0.003f);
*/

    // 모델의 색상을 설정합니다. 여기서는 (0.7, 0.2, 0.3) RGB 색상을 사용합니다.
    glColor3f(0.7, 0.2, 0.3);

    // 쉐이딩 모델을 설정합니다. GL_SMOOTH 는 부드러운 그라데이션 쉐이딩을
의미합니다.
    glShadeModel(GL_SMOOTH);

    // 삼각형을 그리기 시작합니다. 이는 모델의 면을 구성하는 기본 단위입니다.
    glBegin(GL_TRIANGLES);
    for (int i = 0; i < fnum; i++) {
        // 각 면을 구성하는 정점들을 그립니다. mpoint 배열에서 정점의 좌표를 가져와서
사용합니다.
        glVertex3f(mpoint[mface[i].ip[0]].x, mpoint[mface[i].ip[0]].y,
         mpoint[mface[i].ip[0]].z);
        glVertex3f(mpoint[mface[i].ip[1]].x, mpoint[mface[i].ip[1]].y,
         mpoint[mface[i].ip[1]].z);
        glVertex3f(mpoint[mface[i].ip[2]].x, mpoint[mface[i].ip[2]].y,
         mpoint[mface[i].ip[2]].z);
    }
    // 삼각형 그리기를 종료합니다.
    glEnd();
}
```

2-12. 미로배치

이제 display 함수에서 glPushMatrix(), glPopMatrix() 함수를 통해 원하는 객체틀을 회전하고 이동하고 스케일링하여 배치하여 원하는 미로의 구조를 만듭니다. 저희조는 직접 미로의 벽과 객체를 하나씩 직접 변환하여 배치하는 하드코딩 방법을 사용하였습니다.

아래의 그림은 미로를 배치도 입니다. 미로의 벽들을 계산하기 쉽게 배치하기 위해 눈금면에 하나씩 그리면서 미로를 설계하였습니다. 저기에 파란색으로 칠해진 moving wall 은 벽이 가만히 있는 기존의 벽과 달리, 벽의 위치를 계속해서 변경해 주면서 계속 움직이는 객체입니다.

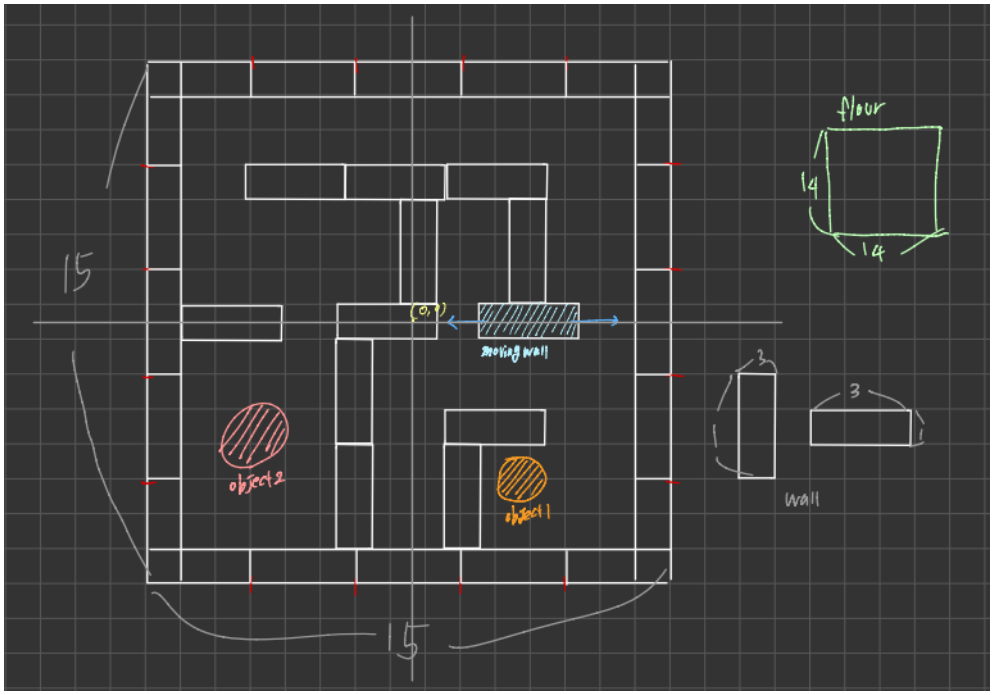


그림 6 미로 배치도

2-13. 미로 안에서 움직이는 벽 만들기

```
// update 함수는 주기적으로 호출되어 애니메이션 상태를 업데이트하는 함수입니다.
void update(int value) {
    // 큐브의 Y 축 위치를 변경하여 큐브를 상하로 이동시킵니다.
    cubeY += cubeU;

    // 큐브가 특정 범위를 벗어나면 이동 방향을 반대로 전환합니다.
    if (cubeY > 5.0 || cubeY < 2.5) {
        cubeU = -cubeU;
    }

    // 화면 갱신을 요청합니다.
    glutPostRedisplay();

    // 16 밀리초(약 60FPS 에 해당)마다 이 함수를 다시 호출하여 애니메이션을
계속합니다.
    glutTimerFunc(16, update, 0);
}
```

벽의 특정 좌표를 계속 업데이트 하려 벽을 움직이게 만드는 함수입니다. cubeY 는 움직이고 싶은 벽의 특정 좌표를, cubeU 는 벽이 움직이는 속도를 나타냅니다. 즉 cubeY 에 cubeU 를 계속해서 더하거나 빼서 벽을 움직이게 할 수 있습니다. 벽이 위치될 위치와 움직여야할 정도를 고려하여 범위를 설정해주고(좌표고 2.5~5), 그 범위를 넘어가면 cubeU 의 부호를 반대로 바꾸어주어 벽이 왔다 갔다 하도록 구현하였습니다. 다음은 display 함수에서 배치한 벽 코드입니다.

```
    glPushMatrix();
```

```
    glTranslatef(cubeY, 1.5f, -0.0f);
    glCallList(WallListID);
    glPopMatrix();
```

위 코드에서 gltranslate 함수에서 벽의 x 좌표에 cubeY 가 들어가 벽의 x 좌표가 계속해서 바뀌어서 벽이 좌우로 왔다갔다 움직이도록 구현했습니다.

2-14. 메뉴를 통해 카메라를 원위치로 돌아가게 하는 기능

추가적으로 화면에 메뉴를 띄우고 reset 버튼을 누르면 다시 원위치로 돌아가는, 카메라 위치를 리셋하는 기능도 구현하였습니다. 먼저 마우스 오른쪽 버튼을 눌렀을 때 메뉴가 생성되도록 하는 코드를 입력해줍니다.

```
// handleMouse 함수는 마우스 이벤트를 처리합니다.
void handleMouse(int button, int state, int x, int y) {
    // 마우스 오른쪽 버튼을 클릭했을 때의 이벤트를 처리합니다.
    if (button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN) {
        // 메뉴를 활성화하고 화면을 갱신합니다.
        isMenuDisplayed = true;
        glutPostRedisplay();
    }
}
```

먼저 조건문에서 button == GLUT_BUTTON && state ==GLUT_DOWN 코드를 통해 오른쪽 마우스 버튼이 눌렸을때를 판별합니다. 그리고 이 조건이 충족하면(마우스 오른쪽 버튼이 클릭되면) isMenuDisplayed 를 true 로 바꾸어줘 메뉴를 표시하도록 합니다.

```
// processMenu 함수는 메뉴 이벤트를 처리합니다.
void processMenu(int option) {
    switch (option) {
    case 1: // '1' 옵션 선택 시 초기 카메라 위치로 이동
        ResetCameraPosition(0);
        break;
        // 이곳에 다른 메뉴 옵션들을 추가할 수 있습니다.
    }

    // 메뉴 선택 후 메뉴를 비활성화하고 화면을 갱신합니다.
    isMenuDisplayed = false;
    glutPostRedisplay();
}
```

다음으로 processMenu 함수로 메뉴에서 선택한 항목에 따라 특정 동작을 수행하는 기능을 넣어줍니다. 즉 메뉴를 눌렀을 때 카메라가 원위치로 돌아가는 함수를 시행시킵니다. 본 코드에서는 1 번 항목이 선택되었을 때, ResetCameraPosition() 함수를 통해 카메라의 위치를 원상태로 돌려줍니다.

```
    glutAddMenuEntry("restart", 1);
```

그리고 메인함수에 위와 같은 코드로 restart 라고 써져있는 1 번 메뉴항목을 추가해줍니다. 이러면 오른쪽 버튼을 누르면 handlemouse 함수에 의해 메뉴가 나오고, 1 번 메뉴항목을 선택하면, resetcameraposition 함수가 시행되면서 카메라가 원점을 돌아갑니다.

```
// ResetCameraPosition 함수는 카메라의 위치와 각도를 초기 상태로 재설정합니다.
void ResetCameraPosition(int value) {
    // 카메라의 X, Y, Z 좌표를 초기 위치로 설정합니다.
    // 여기서 카메라는 원점 근처에 위치하며, 약간 높이 올라가고 약간 뒤로 물러난
위치에 있습니다.
    cameraX = 0.0f;
    cameraY = 1.0f;
    cameraZ = 5.0f;

    // 카메라의 수평 및 수직 회전 각도를 초기화합니다.
    cameraAngleY = 0.0f;
    cameraAngleX = 0.0f;

    // 화면을 다시 그리도록 요청합니다.
    glutPostRedisplay();
}
```

카메라의 위치를 원점으로 돌려주는 함수는 간단합니다. 카메라의 위치와 각도를 초기값으로 재정의 해주면 카메라 초기위치에서 초기 각도로 바라보도록 초기화가 됩니다. 카메라 위치를 원래대로 초기화 하는 함수는 아래와 같습니다.

2-15. Window Reshape 기능

마지막으로 창의 크기를 변경해도 카로세로 비율을 계산해 화면에 보여지는 물체가 일그러지지 않게 해주는 reshape 함수를 구성합니다.

```
// reshape 함수는 창 크기가 변경될 때 호출되어 그래픽스 뷰포트와 투영 매트릭스를
설정합니다.
void reshape(int width, int height) {
    // glViewport 함수는 화면에 그려질 영역의 크기와 위치를 설정합니다.
    // 여기서는 전체 창 크기를 뷰포트로 설정합니다.
    glViewport(0, 0, width, height);

    // glMatrixMode 함수는 현재 작업할 매트릭스 모드를 설정합니다.
    // GL_PROJECTION 은 투영 매트릭스(카메라 뷰) 설정을 위한 모드입니다.
    glMatrixMode(GL_PROJECTION);

    // glLoadIdentity 함수는 선택된 매트릭스 모드의 매트릭스를 단위 행렬로
        초기화합니다.
    // 이는 이전에 적용된 변환을 초기화하는 역할을 합니다.
    glLoadIdentity();

    // gluPerspective 함수는 원근 투영 행렬을 설정합니다.
```

```
    // 이 함수는 시야각(fov), 종횡비(aspect ratio), 근접 평면(near plane), 원거리
        평면(far plane)을 매개변수로 받습니다.
    gluPerspective(45.0f, (float)width / height, 0.1f, 100.0f);

    // 다시 모델뷰 매트릭스 모드로 전환합니다. 이 모드에서는 객체의 변환을
        처리합니다.
    glMatrixMode(GL_MODELVIEW);
}
```

화면의 가로와 세로를 직접 입력하는 것이 아니라 viewport 에서 변경된 가로와 세로의 길이를 width, height 변수를 통해 받아와 투영행열 모드의 가로 세로 비율을 실시간으로 계산하도록 하였습니다.. 때문에 화면의 크기를 줄이거나 늘려도 그리픽이 변형되지 않습니다.

2-16. 구현한 기능 정리

1. 키보드 입력으로 카메라 이동, 이동중에 발걸음 소리 재생
2. 마우스와 키보드 입력으로 카메라 회전
3. 카메라 점프 기능, 카메라 점프시 소리 재생
4. 배경음악 재생
5. 키보드 입력을 통한 카메라 이동 속도 조절
6. 벽과 바닥의 텍스쳐매핑
7. 계속해서 움직이는 3d 모델 객체(움직이는 미로 벽)
8. sor 로 저장한 모델링 파일 불러오기 & 배치
9. 메뉴를 통한 카메라 원위치 이동 기능
10. window reshape 기능

단축키:
Q: 카메라 방향 왼쪽으로 수동 회전
E: 카메라 방향 오른쪽으로 수동회전
1: 카메라 움직이는 속도 +
2: 카메라 움직이는 속도 –
w,a,s,d: 카메라 이동
space: 점프

2-17. 3d maze 시현 사진

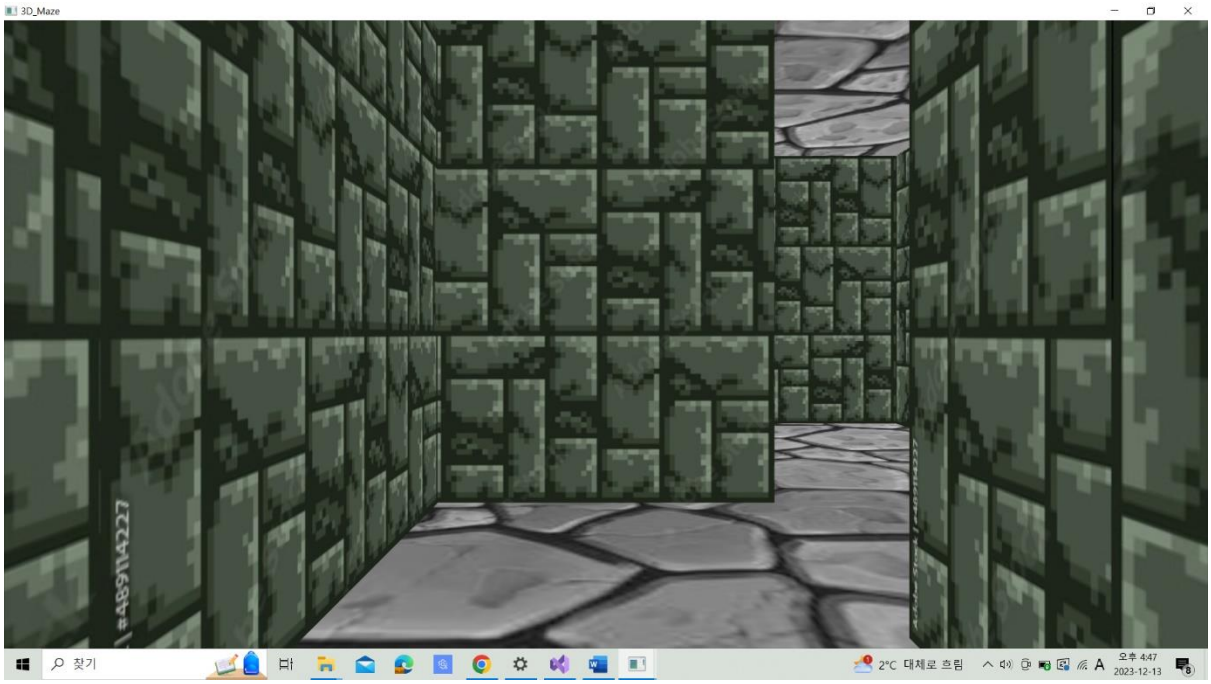


그림 7 3d maze project 시작 화면

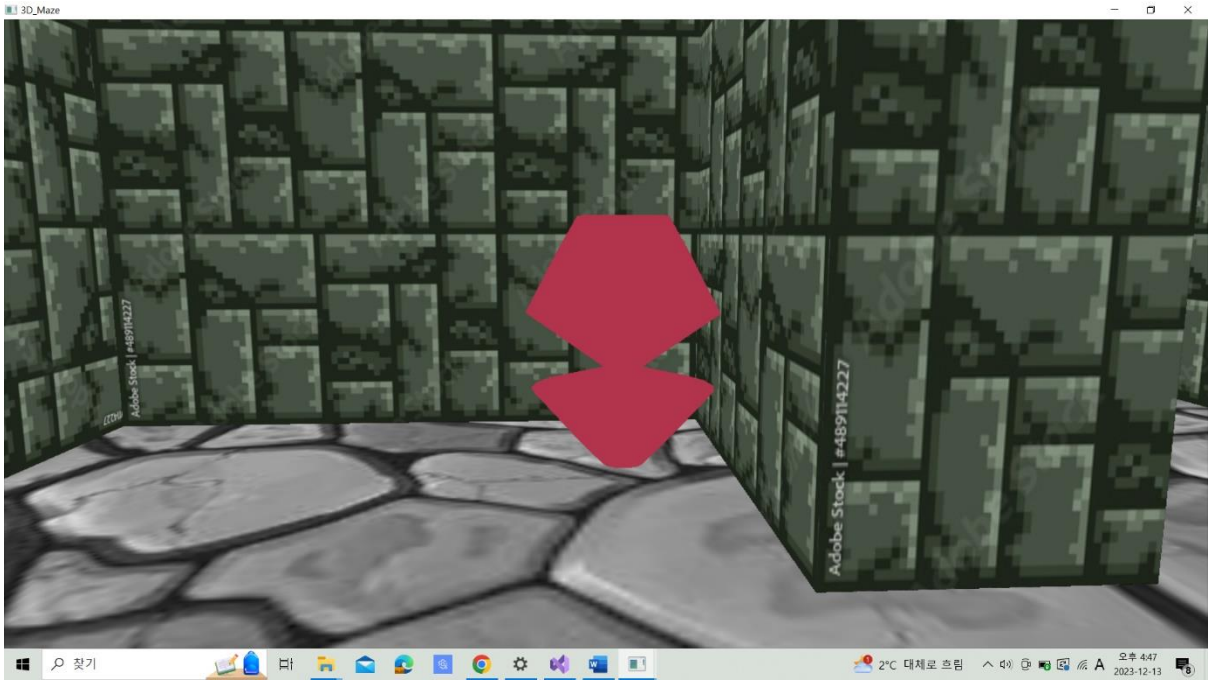


그림 8 sor 모델링 데이터 배치 1

그림 9 3d maze 움직이는 벽

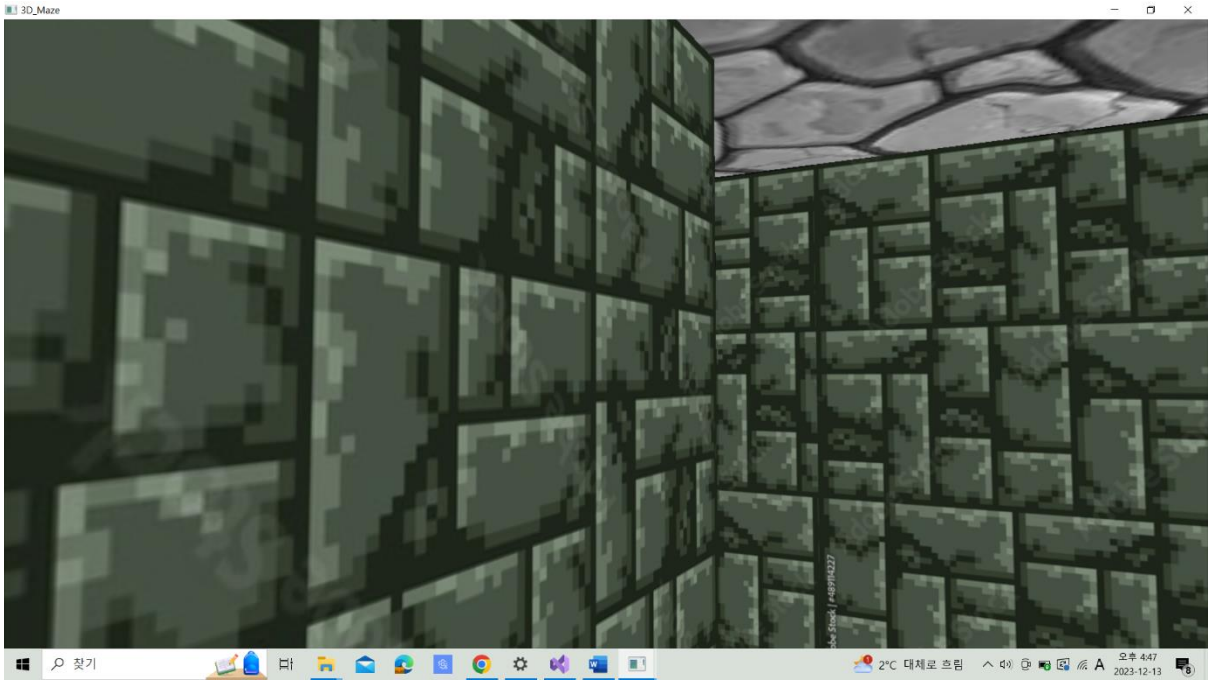


그림 10 점프했을때의 화면

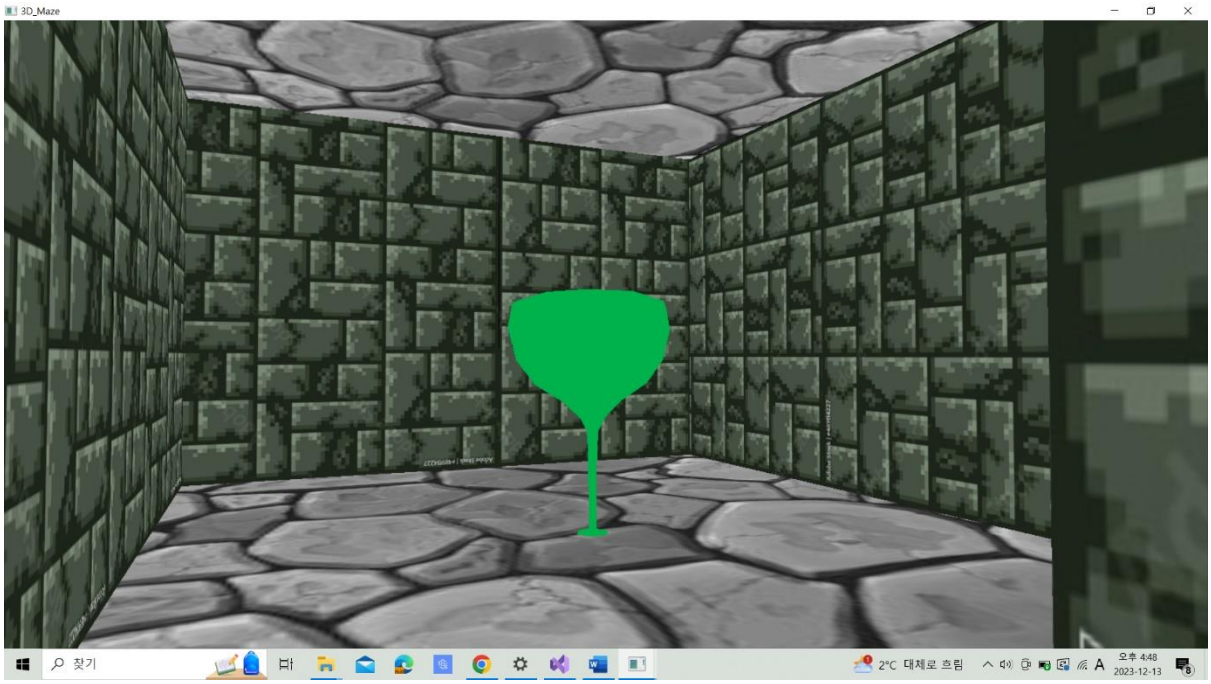


그림 11 sor 모델링 데이터 배치 2

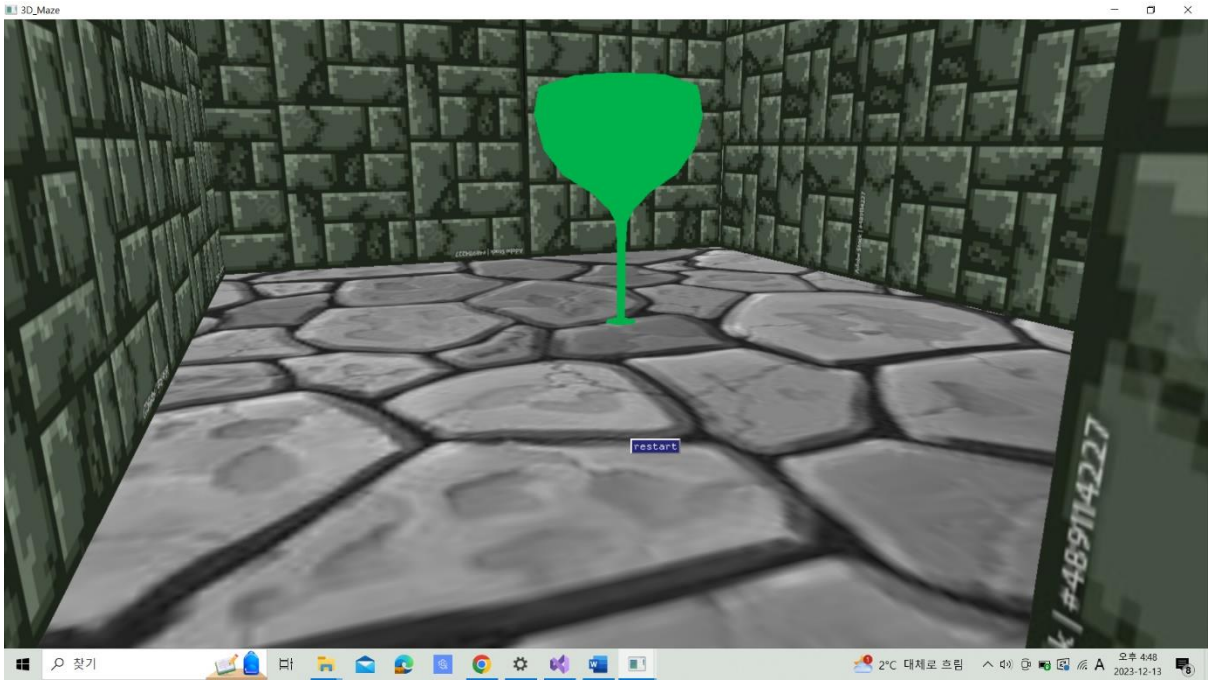


그림 12 restart 메뉴