
컴퓨터그래픽스

과제 3 보고서



세종대학교

제출일	2025.11.22	전공	소프트웨어학과
과목	컴퓨터그래픽스	학번	21011746
담당교수	박상일 교수님	이름	양현석

1. Maze Navigator

a) 문제 분석

주어진 명세에 맞추어 카메라가 미로를 탈출하는 간단한 게임을 만든다. 중점은 다음과 같다.

- 'A' 키를 통한 왼쪽으로 회전, 'D' 키를 통한 오른쪽으로 회전 (좌우 썸 모두 적용)
- 카메라가 벽과 충돌할 때 벽을 뚫지 못하도록 제한 (Collision 처리)
- 'Q' 키를 통한 목표지까지의 최적의 경로 찾기 및 찾은 길을 시각적으로 보여주기
- 'Space Bar' 키를 누르면 카메라가 회전 및 이동을 하여 자동으로 찾은 길을 따라가기

b) 문제 풀이 및 학습 내용

1) 'A', 'D' 키를 통한 회전

기존의 코드는 좌, 우 이동으로 되어 있기에 먼저 기본적인 회전을 할 수 있도록 해준다. 전역 변수로 적용 각도와 더할 각도 상수 offset을 정의하여 idle() 함수 내부에 코드를 추가한다.

```
// Camera Rotate
float g_yAngle = 0.f;
const float g_deltaAngle = 3.f;

// A, D 키를 회전으로 변경
if ((GetAsyncKeyState('A') & 0x8000) == 0x8000)    // if "A" key is pressed : Turn Left
{
    g_isMoving = false;
    g_yAngle += g_deltaAngle;
}
if ((GetAsyncKeyState('D') & 0x8000) == 0x8000)    // if "D" key is pressed : Turn Right
{
    g_isMoving = false;
    g_yAngle -= g_deltaAngle;
}
```

이렇게 각도를 움직이도록 했으니, 이를 적용해야 한다. Display() 함수에서는 현재 1인칭으로 쳐다보는 시점을 나타내야 한다. 현재 바라보고 있는 viewDirection을 y축으로 회전시켜 새로운 방향을 적용한다. drawCamera()에서는 현재 3인칭으로 쳐다보는 시점을 나타내야 한다. 그러나 주어진 코드에서는 회전을 고려하지 않고 카메라를 생성했고, 새로운 코드에서는 카메라가 모양을 유지할 수 있도록 CTM(Current Transform Matrix)를 이용하여 모양을 유지하며 회전할 수 있도록 한다.

```
// Left 화면의 쳐다보는 방향 (변경 사항) - Display() 함수 일부
vec4 rotateViewDirection = RotateY(g_yAngle) * viewDirection;

mat4 ViewMat = myLookAt(cameraPos, cameraPos +
                        vec3(rotateViewDirection.x, rotateViewDirection.y,
                             rotateViewDirection.z), vec3(0, 1, 0));

mat4 ProjMat = myPerspective(45, aspectRatio, 0.01, 20);

g_Mat = ProjMat * ViewMat;
drawScene(false);           // drawing scene except the camera

void drawCamera()
{
```

```

float cameraSize = 0.5;

// 카메라의 앞부분을 함께 이동하기 위해 CTM 사용
mat4 CTM(1.f);

mat4 ModelMat = Scale(vec3(cameraSize));
CTM *= Translate(cameraPos) * RotateY(g_yAngle);
glUseProgram(program);
glUniformMatrix4fv(uMat, 1, GL_TRUE, g_Mat * CTM * ModelMat);
glUniform4f(uColor, 0, 1, 0, 1);
cube.Draw(program);

ModelMat = Translate(viewDirection * cameraSize / 2) * Scale(vec3(cameraSize / 2));
glUseProgram(program);
glUniformMatrix4fv(uMat, 1, GL_TRUE, g_Mat * CTM * ModelMat);
glUniform4f(uColor, 0, 1, 0, 1);
cube.Draw(program);
}

```

이 수정들을 통해 좌화면에서는 시점에 대한 회전이, 우화면의 카메라가 바라보는 방향이 회전을 적용 받게 된다.

2) Collision Resolution

게임을 만들 때 가장 핵심적인 기술인 충돌의 문제를 해결해야 한다. 카메라가 회전되어 있는 상태라면 이를 적용하여 충돌처리를 하면 좋겠지만, 테스트 결과 회전이 되어 있지 않은 상태로 간주하여도 문제가 발생하지 않는다. 따라서 AABB(Aligned-Axis Bounding Box) 충돌 처리를 이용한다. 기본적으로 두 상자 (벽, 카메라)의 범위에 서로가 겹치지 않는지 확인하는 것이 중점이고, 만일 겹치게 된다면 static 물체인 벽이 아닌 dynamic 물체인 카메라를 벽의 충돌 위치에서의 법선 벡터를 통해 밀어낸다. 밀어내는 양은 카메라가 벽을 파고 들어간 거리만큼 진행하며, (파고 들어간 거리 * 법선 벡터) 만큼 카메라의 좌표에 더하여 Collision Resolution을 진행한다.

총 세 개의 함수를 이용한다. 먼저 collisionResolutionHandler()는 이동 거리(offset)을 인자로 받아 카메라를 먼저 offset만큼 이동시킨다. 그 후 collisionDetectionAndResolution() 함수에서 모든 maze를 조사하여 cameraPos와 충돌이 존재하는지를 확인한다. 하나의 블럭과 카메라는 isInCollider() 함수를 통해 AABB 충돌을 확인한다. 각각의 오브젝트의 최대 x, z와 최소 x, z를 비교하는 식으로 진행된다. 비교 식은 다음과 같이 간략화 한다.

```

// Checking - Block 기준
// X Collision
bool xCollision = (collUpperX >= cameraLowerX) && (collLowerX <= cameraUpperX);

// Z Collision
bool zCollision = (collUpperZ >= cameraLowerZ) && (collLowerZ <= cameraUpperZ);

```

만일 여기서 두 값이 모두 true라면 충돌이 있는 것으로 판정한다. 그 이후 카메라가 벽의 어느 방향에서 충돌했는지 판정하여 벽의 법선 벡터를 구한다. 또한 충돌 깊이(겹쳐진 길이)를 확인하는데 이는 충돌이 아닌 범위(최대 거리)에서 각각 성분의 거리를 빼서 구하고, 어느 방향의 충돌이 더 큰지 확인하여 그 방향으로 밀어낸다. 이로써 벽과 부딪히더라도 모든 진행 방향을 막지 않아 벽과 충돌한 상태로도 이동할 수 있다. 아래는 x로 튕겨내는 예시이다.

```

// x로 튕겨내기
if (collLenX < collLenZ) {

```

```

// Camera 가 더 왼쪽
if (cameraPos.x < collCenter.x) {
    wallNormal = vec3(-1, 0, 0);
}
// Camera 가 더 오른쪽
else {
    wallNormal = vec3(1, 0, 0);
}

collisionLen = colLenX;
}

```

collisionLen은 전역 변수로 collisionDetectionandResolution() 함수에서 실제로 밀어낼 수 있도록 한다.

```

if (isInCollider(getPositionFromIndex(i, j), cameraPos))
{
    collisionBlock = &maze[i][j];

    // 충돌의 법선 방향 * 충돌 깊이 만큼 밀어내기
    cameraPos += wallNormal * collisionLen;
    isCollision = true;
}

```

최종적으로 idle() 함수에서 offset을 계산하여 collisionResolutionHandler()를 호출하면, 충돌 처리가 모두 완료된다. 사실 상, 모든 이동을 collisionResoultionHandler()를 통해 진행한다. (이후 A*를 통한 자동 이동에서도 사용)

(3) A* Algorithm 설명

구현을 작성하기 전에, A* Algorithm에 대해 공부한 내용을 먼저 작성하겠다.

A* 알고리즘은 그래프의 최단 경로 문제를 해결하는 알고리즘으로, 다익스트라 알고리즘을 발전시킨 형태이다. 다익스트라 알고리즘은 시작점에서 각 정점에 이르는 최단 경로를, 시작점에서 가까운 정점부터 결정한다. 그러한 이유로 중점에서 멀어지는 방향에 있는 정점들의 최단 경로도 결정하게 되는데 이 계산은 결국 사용되지 않기 때문에 불필요하다. A* 알고리즘은 예상 목표 비용 (Heuristic)을 힌트로 설정하여 불필요한 탐색을 생략하도록 하는 것이 핵심이다.

$$f(n) = g(n) + h(n)$$

- $g(n)$: 출발 Node부터 현재 Node까지의 비용
- $h(n)$: 현재 Node부터 목표 노드까지 예상 비용

→ 결론적으로 $f(n)$ 이 가장 작은 노드를 다음 노드로 설정한다.

위의 식에 따라 A*는 최단 경로를 탐색한다. 이때 사용되는 Heuristic은 사용자가 직접 설정할 수 있는데, 이에 따라 우선순위 큐(최소 힙) 정상 작동 여부 및 성능에 지대한 영향을 끼친다. 일반적으로 다음 세 가지를 사용한다.

- 맨해튼 거리 : $|x_1 - x_2| + |y_1 - y_2|$
 - 수직, 수평 방향으로만 이동할 수 있다고 본다.
- 유클리드 거리 : $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$
 - 대각선으로도 이동할 수 있다고 본다.
- 패턴 데이터베이스 : 미리 계산된 데이터베이스

본 코드에서는 2차원 배열 형태의 미로인 점을 고려하여 맨해튼 거리를 사용했다. (물론 유클리드 거리를 구하여 반올림하여 사용할 수 있는 방법도 가능)

여기서 알 수 있는 점은 Heuristic을 설정할 수 없을 때는 A*를 사용할 수 없다는 것이다. 목표 지점을 명확하게 할 수 없다면, 당연한 것이다. 또 Heuristic을 설정할 때도 주의할 점이 있다. 휴리스틱 비용이 실제 비용과 가까울수록 검색 효율이 높고, 만일 멀다면 다익스트라 알고리즘보다 효율이 떨어지거나 심하면 아예 정답을 찾지 못할 수도 있다. 참고로, 휴리스틱 비용이 실제 비용보다 전반적으로 낮게 설정되어 있다면 바른 답을 찾는 것이 보장된다.

그렇다면, 다익스트라를 A*로 변환하려면 어떻게 해야하는가? 사실 위의 식 $f = g + h$ 를 추가하면 된다고 생각할 수 있지만, 방문 여부를 확인할 수 있는 수단이 필요하다. 이미 최적의 경로로 계산한 경우 다시 방문하지 않도록 하는 것이 중요하기 때문이다. 이는 추가적인 자료구조를 사용하거나, 현재 계산한 비용(우선 순위 큐를 통해 얻은 값)이 미리 계산된 비용보다 크다면 스킵하는 식으로도 구현 가능하다. 다익스트라 알고리즘은 g 를 우선순위로 보았다면, A*는 f 를 우선순위로 보지만, A*에서 g 도 당연히 의미를 갖는다. 또한 A*를 구현할 때 f 는 우선순위 큐 및 배열로 관리, g 는 배열로만 관리하는 등처럼 철저하게 분리해야 한다.

```
// Function Object (Functor)
struct Compare {
    // 함수 호출 연산자를 Overloading
    bool operator()(const std::pair<int, Node>& a, const std::pair<int, Node>& b) const {
        return a.first > b.first;
    }
};

void Dijkstra(const Node & start)
{
    // INF로 초기화 및 쓰레기 값 제거
    for (int y = 0; y < MazeSize; y++) {
        for (int x = 0; x < MazeSize; x++) {
            distArr[y][x] = INT_MAX - 100;
            fromWhereArr[y][x] = { -1, -1 };
        }
    }

    // 시작점은 0
    distArr[start.y][start.x] = 0;

    // 최대힙을 최소힙으로 사용하기 위해 -1을 곱하여 저장 (Priority Queue)
    std::priority_queue<std::pair<int, Node>,
        std::vector<std::pair<int, Node>>,
        Compare
    > pq;

    // Start 저장
    pq.push({ -(0 + abs(endNode.y - startNode.y) + abs(endNode.x - startNode.x)), start });

    // 다익스트라 메인 루프
    while (!pq.empty())
    {
        int f = -pq.top().first;
        Node cur = pq.top().second;
        pq.pop(); // Remove

        int curCost = distArr[cur.y][cur.x];

        // pq에서 꺼낸 값이 distArr보다 좋다면? -> 강 무시
        if (f != curCost + abs(endNode.y - cur.y) + abs(endNode.x - cur.x))
            continue;

        // 현재를 기준으로 주변 4곳 탐색
        for (int k = 0; k < 4; k++)
        {
            // 주변 4곳의 좌표 계산
            int newY = cur.y + coordOffset[k].y;
```

```

        int newX = cur.x + coordOffset[k].x;    // idx를 이렇게 해야 4군데를 다 볼 수 있음

        // Boundary 밖으로 나가지 않았는지 확인 -> 나가면 버려야지 뭐
        if (newY < 0 || newY >= MazeSize || newX < 0 || newX >= MazeSize)
            continue;

        // 벽이어도 버려야지 (이동 못 함)
        if (maze[newY][newX] == '*')
            continue;

        int newCost = curCost + 1;

        int newG = newCost;
        int h = abs(endNode.y - newY) + abs(endNode.x - newX);
        int newF = newG + h;

        // 더 작은 값을 찾았다면 (최소는 distArr를 갱신해야 함)
        if (newG < distArr[newY][newX]) {
            // distArr에는 실제 비용만 저장
            distArr[newY][newX] = newG;
            // 경로를 복원하기 위해 어디서 왔는지를 저장
            fromWhereArr[newY][newX] = cur;
            // -1을 곱하고 push (priority에는 f를 넣어야 함)
            pq.push({ -newF, {newY, newX} });
        }
    }
}
}

```

참고용으로 직접 공부를 위해 작성한 다익스트라 기반에서 A*로의 변형 코드를 첨부하겠다.

하지만, 본 구현에서는 정석적인 A* 알고리즘의 형태로 구현을 하려 한다. 이는 기본적으로 두 가지의 자료구조를 필수적으로 요구한다.

- Open List
 - 지금 탐색할 후보 노드들
- Closed List
 - 이미 최단 비용으로 방문이 확정된 노드들

두 가지 자료구조를 사용하는 이유는 f가 변동되면서 Node가 여러 번 OpenList에 들어갈 수 있다. 그래서 최종적으로 확정된 노드를 ClosedList에 저장하여 불필요하게 다시 방문하지 않도록 하는 것이다. OpenList는 우선순위 큐(최소 힙), ClosedList는 Boolean 2차원 배열로 구현하였다. 이외에도 fCost와 gCost를 저장하는 2차원 배열을 추가적으로 구성한다.

(4) A* Algorithm 구현

먼저 구현의 편리성을 고려하여 Node라는 구조체를 정의한다. 이는 (l, j)를 사용하는 maze를 (x, y)로 더 직관적으로 사용할 수 있게 하고, 여러 operator를 overloading하여 구현의 편리성을 올렸다.

```

/* 간략화 버전 */
struct Node {
    int x;
    int y;

    // 범위 밖을 나가는지 확인
    bool isOutOfBoundary() const;

    // Base에서 뭘 더해야 target으로 가는지 반환
    const Node findOffsetGoToTarget(const Node& target) const;

    // Operator
    bool operator==(const Node& a) const;
}

```

```
Node operator+(const Node& a) const;

friend std::ostream& operator<<(std::ostream& o, const Node& point);
};
```

또 주변 조사를 위해 상하좌우 offset을 저장하는 coordOffset을 만들어 사용한다. gCost, fCost, closedList와 그 외에 경로를 생성하기 위해 어느 노드에서 본 노드로 왔는지를 저장하는 cameFrom과 경로를 저장하는 path를 만든다. mazeSize가 const가 아니기에 동적할당으로 2차원 배열들을 설정한다.

추가적으로, 월드 좌표를 maze 좌표로 환산하는 parseCoordToNode를 생성한다. 이는 getPositionFromIndex를 역으로 진행하며, 혹여나 범위를 나가거나 벽으로 판정되는 것을 막기 위해 주변 좌표로 이동하도록 했다.

가장 먼저 시작점(카메라의 위치)과 목적지(정해진 위치) 좌표를 Node 구조체로 변환하여 저장하고 이를 기준으로 탐색을 시작한다. 가장 먼저 시작점의 f, g 비용을 계산하여 2차원 배열에 저장한 후 openList 역할의 Priority Queue(PQ)에 삽입한다. 이후 Priority Queue가 비거나, 현재 위치가 목적지에 도달하기 전까지 반복한다. PQ에서 f가 가장 작은 Node를 꺼내고, 이를 PQ에서 제거한다. 이 Node가 이미 최소 비용으로 확정된 노드인지 확인하고, 아니라면 이번에 확정한다. 이를 기준으로 상하좌우(한 칸 이동할 수 있는 노드들)를 순회하여 현재 기준 노드를 통한 새로운 이동 비용 g를 계산하고 새로운 비용이 더 싸다면 최선 경로를 수정한다. 또 새로 계산한 노드의 cameFrom을 기준 노드로 변경하고 마찬가지로 fCost도 갱신한다. 이후 이를 PQ에 삽입하여 fCost를 통해 재 탐색을 할 수 있도록 한다. 결론적으로 g, f 모두 최적 거리를 확인하기에 목표 지점에 대한 비용이 커지는 것을 방지한다. 마지막으로, 만일 목적지를 탐색하지 못하는 오류가 발생할 수도 있으므로 탐색하지 못하는 경우에는 false 값을 반환하도록 했다.

openList를 PQ를 통해 구현한다고 했는데, 템플릿 인자로 Compare라는 것을 넣어서 수행했다. 이는 std::sort에 사용되는 비교 기준 함수와 유사한데, sort는 함수명을 넣거나 람다식으로 구현할 수 있는 것과 다르게 PQ에는 본 함수의 리턴 타입이 들어가야 한다. 가장 쉽게 할 수 있는 방법으로는 Function Object(Functor)를 사용하는 방법이 있다. 함수 객체를 만드는 것인데, class 및 struct를 모두 사용할 수 있지만 struct를 사용하였다. 이 구조체의 함수 호출 연산자인 operator()를 Overloading하여 PQ의 비교 함수로써 사용할 수 있도록 했다. 이를 통해 pair의 first인 fCost를 비교한다.

```
/* Priority Queue에서 사용하는 Function Object */
struct Compare {
    bool operator() (const std::pair<int, Node>& a, const std::pair<int, Node>& b) const {
        // 최소 힙을 만들도록 함
        return (a.first > b.first);
    }
};

std::priority_queue <
    std::pair<int, Node>,
    std::vector <std::pair<int, Node>>,
    Compare
>openList;
```

이제 경로를 복원해보겠다. makePath() 함수에서 목표 지점에서 부터 cameFrom을 역으로 읽어 오면 가장 빠른 길을 찾을 수 있다. 이 거꾸로 때문에 처음에는 std::stack을 통해 만들었지만, 디

버깅을 위해 foreach문을 사용하기에는 std::vector가 더 적합하기에 이를 이용하여 path 리스트를 만들었다. 이렇게 하면 begin()에는 목표 지점이 들어가기에 std::reverse() 함수를 이용하여 역순으로 다시 바꿔준다. path 리스트는 전역으로써 사용된다.

```
void makePath()
{
    // path 초기화
    path.clear();

    // 목적지 노드는 cameFrom에 포함되지 않기에 명시적으로 path에 삽입
    Node cur = goalNode;
    path.push_back(cur);

    while (!(cur == startNode))
    {
        cur = cameFrom[cur.y][cur.x];
        path.push_back(cur);
    }

    // 목적지 -> 시작점을 시작점 -> 목적지로 변경
    std::reverse(path.begin(), path.end());
}
```

Q키를 누르면 A*가 작동할 수 있도록 AStarHandler() 함수를 만들어 사용한다. 이 안에서 initializeForAStar(), AStarAlgorithm(), makePath(), postProcessingForAStar() 함수를 호출한다. initializeForAStar()는 fCost, cameFrom 등의 2차원 배열들을 동적할당하고, AStarAlgorithm()는 A* 알고리즘을 수행한다. makePath()는 std::vector 초기화 및 경로를 생성하여 저장, postProcessingForAStar()는 할당된 메모리를 안전하게 해제한다. 또한 경로가 생성되어야 Space Bar 입력을 받을 수 있도록 Boolean 전역 변수 g_movableOnAStar를 두고 이를 true로 만들어준다. 이동처리는 후에 나올 idle()에서 수행한다.

```
void AStarHandler()
{
    initializeForAStar();

    if (!AStarAlgorithm()) {
        std::cout << "Error : AStar Cannot find Path" << std::endl;
        return;
    }

    makePath(); // Global 변수 사용

    // 이동은 idle에서 수행
    g_movableOnAStar = true;

    postProcessingForAStar();
}
```

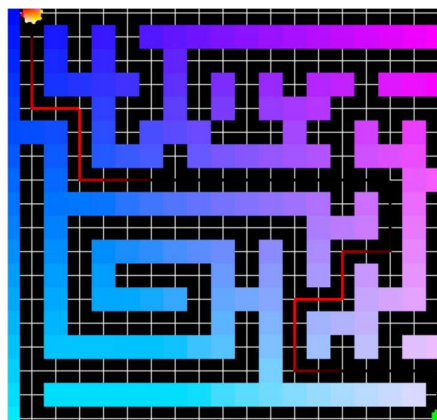
Q키 입력은 다중키 입력을 받지 않아야 하기에 glutKeyboardFunc()로 myKeyboard()를 콜백 함수로 받아, 처리하도록 한다.

4) Path Display

A*를 통해 만든 Path를 주어진 예시와 유사하게 직접 눈에 보이게 해야 한다. 이를 처리하는 함수는 drawPath()로 drawScene()에서 항상 호출된다. 그렇기에, path.empty()가 true인 상황에서는 return 처리를 하도록 했다.

Node들은 maze 좌표로 구현되어 있지만, 그림을 그리기 위해서는 월드 좌표로 그리는 것이 훨씬 그리기 쉽다. 반복문으로 path에서 현 노드와 다음 노드 사이에 하나씩 박스를 그리도록 한다. 각 노드의 좌표를 `parseIndexToCoord()`를 통해 월드 좌표로 환산한다. 참고로 `parseIndexToCoord()`는 기존에 존재하는 `getPositionFromIndex(i,j)`가 (col, row) 형태이기에 이를 더 보기 쉽게 (row, col)로 입력을 받고 `getPositionFromIndex(col, row)`를 리턴해주는 함수이다. 그 후 두 점 사이의 중심을 구하여 그 자리에서 그림을 그린다. 길이는 두 점 사이의 길이이고, 선의 두께 및 선의 높이는 임의로 설정하였다. 박스를 그릴 때는 x, z 방향을 구분하여 그려야 한다. x축 방향의 경우 x의 크기가 두 점 사이의 거리가 되도록, z인 경우에는 z 크기를 두 점 사이의 거리로 두었다.

색상도 빨간색 그라데이션 효과가 나올 수 있도록, Uniform 변수인 `uColor`의 Red 값을 반복문의 iterator `i`와 시간을 관리하는 `g_time`을 적절히 보정하고 `cos()`에 넣어 만들었다.



5) path를 따라 이동

이동을 처리하기 위해 어떻게 할 지 고민하던 중 함수를 통해 만드는 것보다, 상태를 저장할 수 있는 Function Object가 더 좋을 것 같다고 생각했다. 그러나 구현 도중 여러 멤버 함수를 구성하게 되었고, 함수 호출 연산자를 사용하는 것보다 명확한 이름 구분이 필요할 것이라 생각이 들어, struct를 private 없는 class처럼 사용하였다(C++에서는 가능).

A*를 통한 이동은 MoveOnAStar 구조체가 담당한다. 목표 지점(최종 목표지와는 구별되는 개념인 target), 카메라의 시선(방향 벡터), 미로 위에서의 차이 (현 노드 -> target 노드), path 접근용 idx가 있다. 생성자가 따로 없기에 `init()`을 통해 초기화를 진행한다. `init()`은 생성 타이밍이 아닌 Space Bar를 누를 때마다 진행한다.

`init()`에서는 target과 offset을 `path[1]`을 통해 초기화한다. 참고로 `startNode`는 Space Bar 입력 시 재설정된다. 이 때 현재 카메라가 보고 있는 방향을 확인하고 이를 바라보아야 하는 방향으로 보정하는데 초기화와 동시에 이루어져야 하기 때문에, `while(rotate())`의 형식으로 동작한다. 그러나 만일 `path.size()`가 1보다 작거나 같다면 이동의 의미가 없기에 false를 반환하여 이동이 수행되지 않도록 했다.

이동은 `move()`에서 이루어진다. 먼저 path가 생성된 상태인지 확인하고 아니라면 return한다. 그 후 만일 회전을 수행해야 하는 순간이라면 이동을 하지 않고, 회전을 모두 수행한 후 이동을 해

야할 것이다. 따라서 while(rotate())을 수행한다. 위와 다른 점은 move()는 idle()에서 호출되기 때문에 Sleep()을 따르는 것이다. 즉, FixedUpdate()를 따른다고 볼 수 있다. 만일 회전이 끝났다면, 카메라가 현재 바라보는 방향으로 이동한다. 이동을 수행하다가(idle()에서 반복 호출) 만일 목표지에 도달한다면 목표지를 다음 노드로 수정하고 offset도 함께 계산한다. 만일 목표지에 도착했다면, 이동을 종료한다.

```
void move()
{
    if (!g_movableOnAStar) return;

    if (rotate())
        return;

    // 현재 바라보는 방향으로 이동 (속도 * 방향 벡터)
    collisionResolutionHandler(cameraSpeed * getOrientFromOffset());

    if (isSimilar(cameraPos, parseIndexToCoord(target.x, target.y))
    {
        idx++;

        // 이동 종료
        if (idx >= path.size()) {
            g_isMoving = false;
            g_movableOnAStar = false;
            return;
        }

        offset = target.findOffsetGoToTarget(path[idx]);
        target = path[idx];
    }
}
```

아래 코드는 idle() 내부에서의 move()의 호출 부분과 myKeyboard()의 Space Bar 입력 부분이다.

```
if (g_isMoving)
    moveHandler->move();

case ' ':
    cameraPos = getPositionFromIndex(startNode.y, startNode.x);
    if (!g_movableOnAStar) break;
    if (moveHandler) delete moveHandler;
    moveHandler = new MoveOnAStar;

    // Path가 너무 짧다면
    if (!moveHandler->init()) break;

    g_isMoving = true;
    break;
```

6) path를 이동하던 중 회전

이번에는 MoveOnAStar의 회전 처리에 대한 구현이다. 먼저 회전을 하기 전에 어느 방향으로 돌아야 하는지를 알아야 한다. 다양한 방법이 있지만, 게임 분야에서 많이 사용하는 CCW(Counter ClockWise)라는 방법을 사용하고자 한다(물론 멀티미디어 프로그래밍에서도 배움). 시계 반대 방향으로 회전하는지를 판단하는 방식이며, 만일 반대 방향이라면 +1, 정방향이라면 -1, 동일하다고 판정이 된다면 0을 출력하도록 했다.

그러나 2차원에서의 CCW와 3차원에서의 CCW에는 차이가 있다. 3차원에서는 외적을 구했을 때

윗, 아래 방향을 확인해야 한다는 것이다. 따라서 회전 축인 Y축과 카메라 방향과 목표 방향의 외적을 내적하여 방향을 추가적으로 확인하도록 한다. 즉 이렇게 구한 값이 CCW를 판단한다. 원래 기본적으로 CCW는 0을 기준으로 판단되어야 한다. 그러나, 각도가 꽤 크게 도는 것과 float 연산의 오차 값, 내적 외적의 오차 값을 생각했을 때 0보다는 조금 크거나 작은 수를 기준으로 두어야 한다는 것을 알았다(이후 시행 착오에서 자세히 기술). 또한 만약에 카메라 방향과 목표 방향이 정확히 반대라면 뒤집힌 상태에서 움직이는 현상을 목격했다. 따라서 외적을 통해 -1에 가깝다면 무조건 회전시켜서 init()단계에서 보정될 수 있도록 했다(이 현상은 처음 시작할 때만 발생). 아래는 완성된 코드이다.

```
int counterClockWise()
{
    // Normalize 하여 방향 벡터로 비교
    vec3 base = normalize(getOrientFromOffset());
    vec3 cam = normalize(cameraOrientation);

    // 아래의 ccw 는 문제가 있음 정확히 반대 방향이면 그대로 움직이는 문제 발생
    float dotVal = dot(cam, base);

    // 180 도 반대 방향 (dot ~ -1)
    if (dotVal < -0.9999f) {
        // 내적이 반대 방향을 시사한다면, 무조건 회전 (init 에서 수행될 예정)
        return 1;
    }

    // 3 차원에서의 CCW
    // cross(cam, base)가 방향이 정해지고 y 과 내적하면 방향을 추적할 수 있음
    float ccw = dot(vec3(0, 1, 0), cross(cam, base));

    // 원래는 0 을 기준으로 구분하지만, 각도가 크기 때문에 빙글빙글 도는 문제 발생을 해결하고자 도입
    const float standard = 0.0001f;
    if (ccw > standard) return 1;
    if (ccw < -standard) return -1;
    return 0;
}
```

방향을 판정했으니, 이제 회전을 구현하겠다. CCW의 반환 값이 0이라면 회전을 수행하지 않고, 1 이나 -1이라면 이를 g_deltaAngle(회전 상수)에 곱하여 g_yAngle에 더한다. 그러지는 것은 g_yAngle을 통해 자동으로 수행될테니 cameraOrientation만 이에 맞추어 변경해주면 된다. 이는 vec3 -> vec4 -> vec3의 단계로 대입해주었다. 이는 CCW가 0이 될 때까지 move() 내에서 반복된다.

```
bool rotate()
{
    int ccw = counterClockWise();

    // 동일하다고 판명이 난다면, CameraOrientation 을 Offset 의 Orientation 으로 변경 (회전 끝)
    if (ccw == 0) {
        cameraOrientation = getOrientFromOffset();
        return false;
    }

    g_yAngle += ccw * g_deltaAngle;

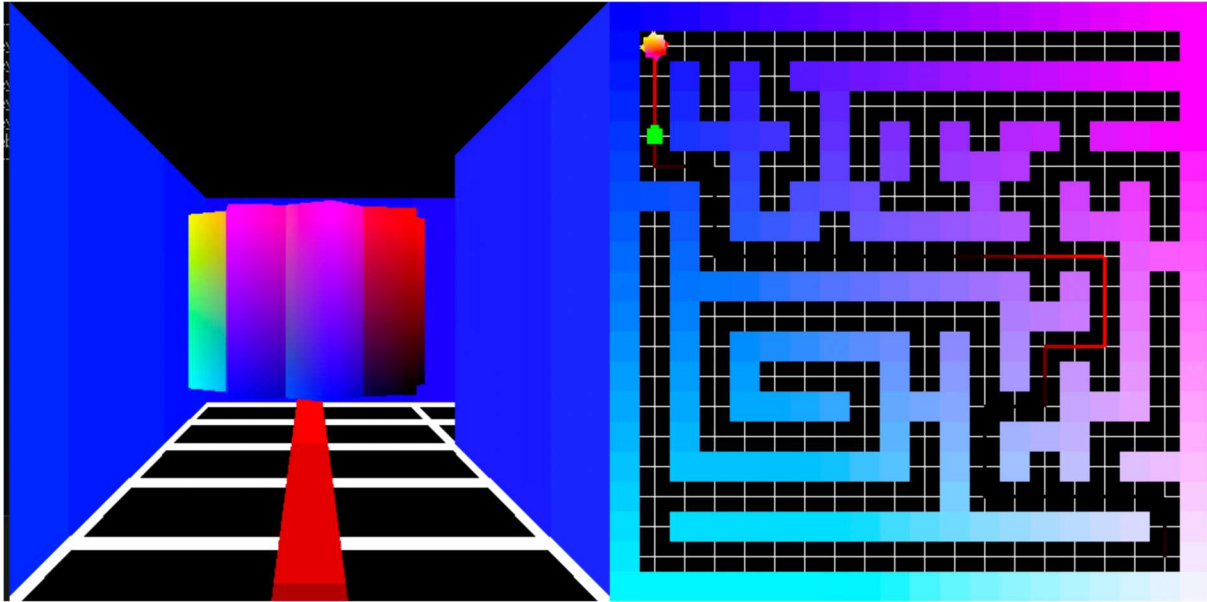
    vec4 cR4 = vec4(cameraOrientation, 1);
    cR4 = RotateY(ccw * g_deltaAngle) * cR4;

    cameraOrientation = vec3(cR4.x, 0, cR4.z);

    return true;
}
```

참고로 getOrientationFromOffset()은 offset 값을 월드 좌표계로 만들고, 원점을 월드 원점으로 이

동시켜 normalize하여 방향 벡터로 반환한다.



c) 시행착오 및 추가적인 학습 사항

많은 기능을 구현해야 하는 과제이기에 그에 맞게 시행착오도 많이 존재했습니다. 위에서 언급한 시행 착오에 대한 구체와 대표적인 시행 착오 그리고 추가적인 학습 내용에 대해 정리해보겠습니다. 추가적으로 이해를 돕기 위해 실수한 코드를 함께 기재해두었습니다.

1) Collision Resolution

처음 충돌 처리를 했을 때는 Hard한 방식으로 진행했었습니다. 충돌이 감지되었을 때 아예 벽 방향으로의 이동을 금지시키는 것이었습니다. 그러나 제가 생각했던 것은 카메라가 벽 방향으로 이동할 때 그 벡터를 벽을 미끄러져 이동할 수 있도록 Projection시키는 방식이었습니다. 하지만, 이를 구현하는 것은 너무 복잡하고 어려워 그와 유사한 효과를 낼 수 있도록, 들어온 만큼만 벽 밖으로 밀어내는 식으로 진행했습니다. 즉, 밀어낸 만큼을 제외하고는 이동할 수 있게 되는 것입니다. 그를 구현하기 위해서 벽의 법선 벡터와 카메라의 이동 방향을 조합하여 카메라가 진행할 수 있도록 했습니다. 그러나, 문제는 이렇게 밀어내었을 때 주변 벽과 다중 충돌이 있을 수 있다는 점이었습니다. 기존 코드에서는 다중 충돌에 대한 처리가 제대로 되지 않았고, 충돌처리는 한번만 발생하여 결국 벽을 뚫고 지나가는 문제가 발생했습니다. 이를 해결하기 위해 코드를 짜다 보니, 너무 코드가 복잡하고 작동도 제대로 되지 않았습니다. 하지만 결국에는 Collision 체크만 여러 번 반복하면 된다는 것을 깨달았습니다. 아래는 문제가 되었던 코드입니다.

```
// Soft Resolution
void collisionResolution(const vec4 & offset)
{
    vec3 preCameraPos = cameraPos;
    vec3 offsetVec3 = vec3(offset.x, offset.y, offset.z);

    cameraPos += offsetVec3;

    // 진행방향 및 벽의 위치를 고려하여 Projection 되도록 세팅 + 일정 부분 떨어져야 함.
    if (collisionDetection(cameraPos)) {
        // 튕겨나갈 vec3 찾기

        // 밀어내기
        cameraPos += wallNormal * collisionLen;

        // Sliding
        vec3 resolutionVec3 = offsetVec3
            - dotProduct(offsetVec3, wallNormal) * wallNormal;

        // 바로 옆 벽에 닿으면 뚫리는 문제 방지 - 의도적으로 10 번까지 체크
        vec3 resolutionBuffer = resolutionVec3;
        for (int i = 0; i < 10; i++) {
            if (collisionDetection(cameraPos + resolutionBuffer, false)) {
                cameraPos += wallNormal * collisionLen;

                resolutionBuffer = offsetVec3
                    - dotProduct(offsetVec3, wallNormal) * wallNormal;
            }
            else break;
        }
    }
}
```

2) 좌표 체계

Maze의 index와 월드 좌표계는 서로 뒤바뀌어 있다는 것에서 너무 복잡하게 생각하게 되었습니다. 결국에는 Node 구조체 내부에 toggleXY()이라는 기괴한 함수를 만들어 사용하게 되면서, 좌표 체계가 위태로울 정도로 모호해지게 되었습니다. 그러던 중 주어진 getPositionFromIndex() 함수를 랩핑하는 함수를 하나 더 만들어 사용하면 Node 따로 월드 좌표 따로 사용할 수 있다는 것이 좋아졌습니다. 그렇게 하여 parseCoordToNode()와 parseNodeToCoord()를 만들어 좌표 체계를 더 탄탄하게 설정할 수 있었습니다.

```
// x, z 축 반전을 고려해 다음과 같이 사용
void toggleXY() {
    int temp = x;
    x = y;
    y = temp;
}
```

3) Path 그리기

2번의 시행 착오인 좌표 체계에 대한 오해와 오류 때문에, Path를 그리는 것 또한 기괴한 코드가 되었습니다. 원래대로라면 World 좌표계를 기준으로 그림을 그리는 것이 맞는데, maze의 좌표를

기준으로 그림을 그리다보니 제대로 그려지지 않을 뿐더러 코드 또한 이해하기가 어려웠습니다. 이 시행 착오를 통해 어느 상황에서 어떤 좌표계를 쓰는 것이 적합하며, 또한 좌표 체계가 얼마나 큰 의미를 갖는지 몸소 느낄 수 있었습니다. 아래는 Node를 기반으로 Path 코드를 작성해보았던 것입니다. 보시다시피 Translate에 무턱대고 Node의 좌표를 넣으려 한 것이 가장 큰 문제였습니다. 현재는 월드 좌표로 두 노드의 중심에서 그리도록 수정했습니다.

```
// Path를 시각적으로 그리기 위함
void drawPath()
{
    // Path가 비어있다면 Return
    if (path.empty()) return;

    float n = 40;
    float w = MazeSize;
    float h = MazeSize;

    // 그리드 한 칸의 크기
    float divW = w / n * 2;
    float divH = h / n * 2;

    // 기준 노드 확인 (시작점부터)
    Node& baseNode = startNode;
    const float pathWidth = 0.1f;
    const float moveDiv = divW / 2; // draw는 중앙에서 그림

    for (std::vector<Node>::iterator it = path.begin(); it != path.end(); it++)
    {
        auto node = (*it);

        Node offset = baseNode.findOffsetGoToTarget(node);

        // xz축이 바뀌어 있는 것을 고려
        offset.toggleXY();

        mat4 m(1.f);

        if(g_showPathLog)
            std::cout << "Base : (" << -w / 2 + baseNode.y * divH << ", " << -(float)h / 2 +
baseNode.x * divW << ")"
                << "->" << node << " Offset : " << offset << std::endl;

        int moveDiv2 = (offset.x < 0 || offset.y < 0) ? 0 : 1;

        // Node : 0 ~ 19의 좌표
        if (offset.x == 0)
        {
            m = Translate
                (-h / 2 + baseNode.y * divH + moveDiv, -0.4, -w / 2 + (baseNode.x +
moveDiv2) * divW)
                * Scale(pathWidth, 0.02, abs(offset.y));
        }
        else
        {
            m = Translate
                (-h / 2 + (baseNode.y + moveDiv2) * divH, -0.4, -w / 2 + baseNode.x *
divW + moveDiv)
                * Scale(abs(offset.x), 0.02, pathWidth);
        }

        glUniformMatrix4fv(uMat, 1, GL_TRUE, g_Mat * m);
        glUniform4f(uColor, 1, 0, 0, 1);
        cube.Draw(program);

        baseNode = node;
    }

    g_showPathLog = false;
}
```

4) CCW 오류

위에서 구현한 CCW에서 기준으로 0으로 두지 않은 것과 카메라와 목표 방향이 정확하게 반대가 되어서 발생한 문제에 대해서 기술하겠습니다. 먼저, 기준으로 0으로 두지 않은 것에 대한 것입니다. 디버깅을 수행하던 중 회전을 했을 때 한 구간에서 무한 루프가 발생했습니다. 사유는 CCW값을 print하여 알 수 있었습니다. 아주 작은 양수 값과 아주 작은 음수 값을 왔다 갔다 하는 것이었습니다. 즉, +1과 -1가 무한히 반복되며 반환된다는 것이었습니다. 각도 상수를 변경하여도 진행해보았지만, 효과는 미미했습니다. 따라서 float 값이 오류를 갖는다는 것을 생각했고, 이를 보정하기 위해 isSimilar() 함수(실수를 비교하기 위해 만든 함수)와 같이 standard라는 값을 정하여 0에서 딱 비교하지 않도록 했습니다. 이를 통해 해결할 수 있었습니다.

다음은 목표 방향이 카메라 방향에 정확히 반대를 향해 있을 때, 그 상태로 이동하는 것이었습니다. 이는 매우 특수한 상황이지만 방향을 앞인지 뒤인지 구분할 수 없기에 발생한 문제였고, 따라서 내적을 계산하여 -1에 가까운 값이라면 반대 방향이기에 아무 각도를 돌리도록 했습니다. 즉, 아무 각도를 돌리게 되면 결국에는 회전이 이루어져야 하는 상태가 되기 때문에, 정확한 방향을 볼 수 있게 됩니다.

5) GetAsyncKeyState() 공부

기존에 OpenGL을 이용하여 프로그램을 만들 때는 glutKeyboardFunc()를 이용하는 함수를 사용했었습니다. 이번에는 GetAsyncKeyState()라는 함수를 사용하여 이동 및 연속 키 입력을 구현했습니다. GetAsyncKeyState()는 매 프레임 호출하여 키가 눌러있는지를 확인하는 폴링 방식으로 구현되어 있습니다. 이는 이벤트 방식으로 키 입력을 처리하는 glutKeyboardFunc()와 차이가 있습니다. glutKeyboardFunc()는 키가 눌리면 이벤트가 발생하여 이를 처리하는데, 눌리고 있는 상태여도 최초 1회의 입력만 이벤트로 감지한다는 것입니다. 반대로 GetAsyncKeyState()는 마치 언리얼 엔진의 tick()에서 입력을 검사하는 것과 같이 매 프레임 검사하기에 연속적 키 입력을 받을 수 있습니다. 본 과제와 같은 게임의 이동과 같은 로직을 구현하기에는 GetAsyncKeyState()를 사용하는 것이 적합하다는 것을 알 수 있었습니다. 그리하여, 이동 및 회전에 대한 입력은 GetAsyncKeyState()로 받고, 'Q', 'Space Bar' 키는 연속적 입력을 받으면 오히려 문제가 생길 수 있기에 glutKeyboardFunc()를 통해 이벤트가 호출될 수 있도록 했습니다.

평소 게임 개발 및 C++에 관심이 많은 제게 C++ 및 OpenGL을 통해 벡터를 직접 조작하고 Collision 처리 및 A* 알고리즘을 구현할 수 있는 매우 좋은 기회였습니다. 게임 엔진에서 제공하는 충돌 처리 등이나 최단거리 계산(NavMeshAgent in Unity)을 사용해보았지만, 실제로 어떻게 구현되어 있는지에 대해서는 깊게 생각해볼 기회가 없었던 것 같습니다. 또한 프로그램 길이가 길어지며 어떻게 설계할 지에 대해서도 고민해볼 수 있는 기회였던 것 같습니다. 좋은 과제를

내주셔서 감사합니다.

d) 참고 자료

- [그림으로 이해하는 알고리즘 \(이시다 모리테루 외 1인, 길벗\)](#)
- [A* 알고리즘 \(NewCodes, Youtube\)](#)
- [A star 알고리즘 \(그리타GRITA, Youtube\)](#)
- [CCW\(Counter-Clockwise\) 알고리즘 \(JoonDev, Tistory\)](#)
- [GetAsyncKeyState 함수 사용법 \(코딩팩토리, Tistory\)](#)
- [함수 객체\(Functor\) \(팡트루야, Tistory\)](#)

이상입니다.