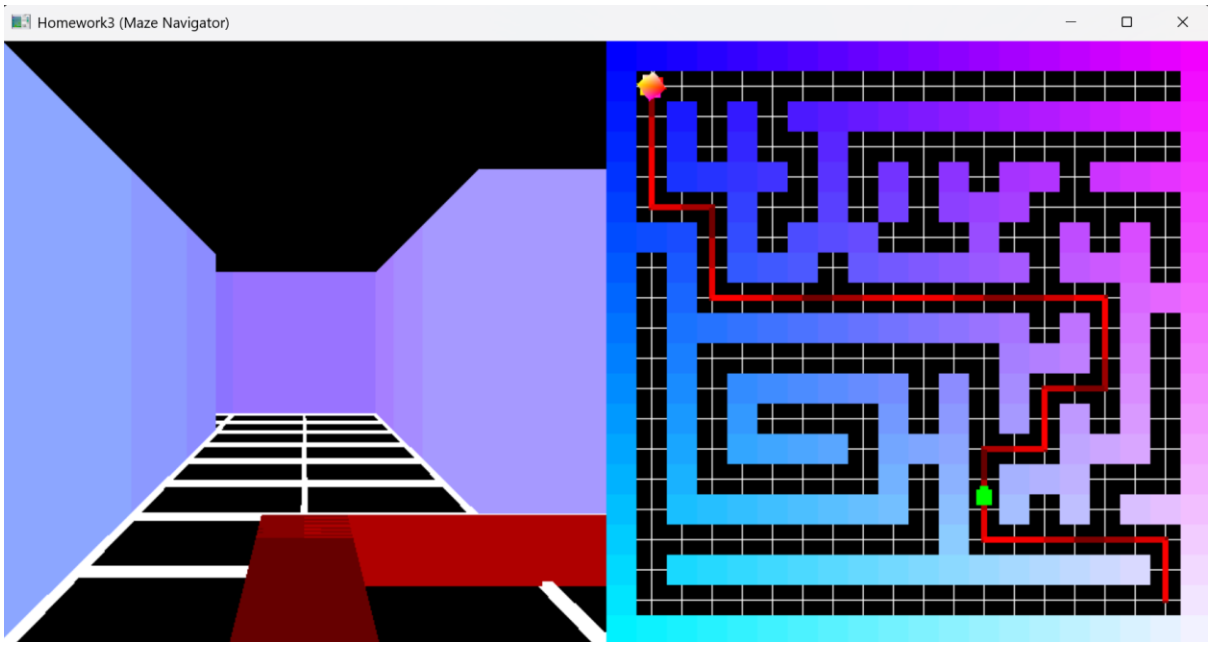
Homework#3: Maze Navigator 만들기에 관한 보고

21011794 황재동(2025.11.19)

1. 개요:

본 과제는 3D 가상 환경에서 1인칭 시점의 카메라를 제어하여 미로를 탐색하고 알고리즘을 통해 최단 경로를 찾아내는 Maze Navigator를 구현하는 것을 목표로 한다. 기존 키 입력 시스템을 보완하여 A, D키를 입력하면 카메라가 방향에 맞게 회전하게 한다. 그리고 단순히 키보드 입력을 수정하는 것이 아닌 벽과의 물리적인 충돌처리(Collision Detection)와 A-star 알고리즘을 이용하여 경로를 탐색하고 탐색한 경로를 기반으로 최단거리를 이동하게 한다.



2. 구현 방법 구상

2-1. 사용자 조작 개선(Key input 수정)

WASD키가 모두 이동만하게 구현이 되어 있어 AD키를 누르면 카메라가 회전하게 수정해야 한다. 매우 간단하게 기존에 있는 viewDirection 벡터 변수에 원하는 회전값 행렬을 곱해주면 구현이 될 것이다. 대신 카메라를 그리는 부분에서 렌즈와 몸통이 따로 그려지기 때문에 해당 부분만 올바르게 수정하면 된다.

2-2. 물리적 상호작용 구상

카메라가 벽을 뚫고 지나가게 되면 미로 찾기의 의미가 없고 키 조작만으로 벽을 고의적으로 피해가는 것도 별로 좋지 못하다. 그래서 카메라가 벽에 닿으면 더 이상 입력 방향으로 진행하지 못하게 막아야 한다.

이동 명령이 발생할 때마다 다음 위치의 유효성을 검사하고 카메라가 벽과 수직으로 부딪히면 그냥 멈춰도 자연스럽겠지만 비스듬하게 닿아 있다면 그렇지 않다. 보통의 게임들을 봐도 플레이어가 벽에 비스듬하게 닿아 있으면 벽면을 타고 미끄러지듯이 움직인다. 키입력을 X, Z축으로 나누고 X, Z축의 값을 따로 추가해주면 구현할 수 있을 것이다. 또 카메라는 부피가 있기 때문에 이것을 고려하여 약간의 여유 공간을 두어야 조금 더 자연스러울 수 있다.

2-3. A-star 알고리즘을 통한 경로 탐색 및 따라가기

시작지점에서 목표지점(C에서 G까지)까지의 최단 경로를 계산하기 위해 다익스트라 알고리즘이 아닌 현재지점부터 목표지점까지의 예상 비용까지 고려하는 A-star 알고리즘을 사용할 것이다. 이를 통해 탐색된 경로는 시각적으로 표시하고, 스페이스바를 누르면 카메라가 해당 경로를 따라 자동으로 주행하는 기능도 추가한다. 이때 카메라가 모서리에서 회전할 때 딱딱하게 도는 것이 아니라 부드럽게 회전하여 이동하도록 보간 처리를 하면 자연스러울 것이다.

3. 구현 단계

3-1. 시점 회전 및 카메라 모델

먼저 시점 회전을 구현하기 위해 idle()함수 내에서 'A', 'D' 키 입력 받는 부분을 회전으로 수정했다. 키가 눌리면 RotateY() 함수를 사용하여 viewDirection 벡터를 Y축 기준으로 좌우 회전하게 했다. 추가적으로 고려해야 될게 상단 탑뷰 녹색 카메라 모델도 시선방향과 일치해야 한다. 이를 위해 drawCamera()함수에서 atan2() 함수를 사용해서 현재 시선 벡터를 각도로 변환했고 이때 구한 각도를 RotateY() 함수에 적용하여 카메라 모델이 실제 플레이어가 바라보는 방향으로 회전하도록 했다.

3-2. 부드러운 충돌 처리

벽과의 충돌을 감지하기 위해 boolean을 반환하는 isWall() 함수를 구현했고 내부에서는 월드 좌표를 받아 그리드 인덱스로 변환한 뒤 해당 미로 배열 값이 벽인지 감지한다. 이때 카메라가 벽에 완전히 닿기 전에 멈추도록 하기 위해 padding 변수를 사용하여 카메라 중심점으로부터 일정거리 떨어진 곳에서 위치를 검사하게 했다. 또한 자연스러운 움직임(벽에 부딪혀서 이동할 때 벽면 따라 미끄러짐)을 위해 함수 이동 로직에서 벡터 연산을 분리했다. 이동하려는 벡터를 한번에 더하는 방식이 아니라 X축 이동과 Z축 이동을 각각 독립적으로 검사한다. 이렇게 함으로써 X축 이동에 벽에 막히더라도 Z축이 뚫려있다면 이동을 허용하여 벽을 미끄러지듯이 이동할 수 있게 됐다.

3-3. A-star 알고리즘 구현

미로의 최단 경로 탐색을 위해 FindPath() 함수에 A-star 알고리즘을 구현했다. 먼저 좌표(x,y), 비용(G,H,F) 등을 관리하기 위한 Node 구조체를 정의했다. 탐색의 효율성을 높이기 위해 GetHeuristic()함수에서는 유클리드 거리가 아닌 격자 이동에 최적화된 맨해튼 거리 공식을 사용하여 목표지점까지 예상 비용을 계산했다(H). 탐색 루프에서는 searchNodeList에서 총비용(F)이 가장 낮은 노드를 우선적으로 꺼내고 상하좌우 4방향의 이웃노드를 검사한다. 이웃노드가 벽이 아니고 방문하지 않았거나 더 짧은 경로가 발견되면 부모노드를 갱신하고 리스트에 추가하는 방식으로 목표지점까지의 최적 경로를 찾아낸다. A-star 알고리즘에 대한 자세한 설명은 나중에 이어서 한다.

3-4. 경로 시각화

탐색된 경로는 DrawPath() 함수를 통해 시각화 했다. 단순히 점을 찍어 진행하는 것이 아닌 두 노드 사이의 중간 지점을 구하고 Scale을 눌려 끊어지지 않고 이어진 선분처럼 보이게 했다. 여기에 시각적으로 예뻐 보이기 위해 빛이

흐르는 것처럼 했다. Sin 함수에 시간 변수 g_time과 경로의 인덱스를 조합하여 빛이 시작점에서 목표지점으로 흐르듯이 보인다.

3-5. 자동 주행 시스템(현재 지점부터 목표지점까지)

자동 주행은 idle()함수에서 스페이스바 입력을 감지하여 bAutoRun 상태를 토글하는 방식으로 제어했다. 자동으로 이동하는 중에 스페이스바를 다시 누르면 정지하고 수동 키 입력을 다시 받을 수 있고 스페이스바를 또 누르면 직전 경로를 따라 자동으로 이동한다. 주행 중에는 카메라가 다음 경로지점을 향해 부드럽게 회전하도록 했는데 목표 방향 벡터와 현재 시점 벡터 사이를 선형 보간하여 방향이 급격하게 직각으로 돌지 않고 조금씩 회전하여 부드러운 곡선을 그리며 이동하도록 했다.

4. A-star Algorithm

미로와 같은 격자 환경에서 최단 경로를 효율적으로 찾기 위해 A-star 알고리즘을 코드에 상세하게 구현했다. 이 알고리즘은 시작노드에서 목표 노드까지의 경로 중 최소 비용을 찾는 경로를 탐색한다.

4-1. Node 구조체 정의

각 격자의 상태를 관리하기 위해 아래의 정보를 담고 있는 Node 구조체를 정의했다

- 좌표(x,y): 2차원 배열상의 인덱스
- 비용 (G, H, F): 경로 비용 평가를 위한 변수
- 부모 포인터(parent): 현재 노드에 도달하기 직전에 거쳐온 노드를 가리키는 포인터로 목표 도달 후 경로를 역추적 하는데 사용된다.
- 상태 플래그들(visited, isSearch): 중복 탐색을 방지하기 위한 방문 여부 및 대기열 포함

4-2. 비용 계산 함수(F = G + H)

G(Goal Cost): 시작점으로부터 현재 노드까지 이동하는데 소요된 실제 비용이다. 인접한 격자로 이동할 때마다 비용을 1씩 누적한다. (G next = G current + 1.0)

H(Heuristic Cost): 현재 노드에서 목표지점까지 남은 예상비용이다. 미로는 그리드로 구성되어 있어 맨해튼 거리(Manhattan Distance)사용이 적절하다고 판단하여 이를 사용했다. 수식은 매우 간단하다.

H = |x current − x goal| + |y current − y goal|

F(Final Cost): F = G + H로 정의할 수 있으며 최종 비용이다.

4-3. 탐색 과정

1. 초기화: 시작노드의 G,H,F를 계산하고 searchNodeList에 추가
2. 노드선택: searchNodeList에 있는 노드 중 F 비용이 가장 저렴함 노드를 curent로 꺼낸다.
3. 목표 검사: 꺼낸 노드가 목표지점이면 탐색을 종료하고 경로 추출
4. 이웃 탐색: 현재 노드의 상하좌우 검사. 이때 맵을 벗어나거나 벽 또는 이미 방문된 노드는 무시한다.
5. 비용 갱신: 새로운 경로가 기존에 알고 있던 경로보다 짧거나 해당 노드를 처음 발견한 경우
   A. 이웃의 부모를 current로 설정
   B. G, F 비용을 갱신하고 searchNodeList에 추가
6. 위 과정을 리스트가 비워질때까지 계속 진행

4-4. 경로 추출

목표지점에 도달하면 Current 노드에서 시작하여 parent 포인터를 따라 시작점까지 거슬러 올라간다. 이 과정에서 경로를 목표지점에서 시작지점까지의 순서로 추출되어서 실제 로직에 적용하기 위해 이를 뒤집어야 한다.

5. 시행착오 및 해결과정

5-1. 카메라 모델의 역회전 문제

1인칭 시점은 정상적으로 회전하는데 탑뷰의 녹색 카메라 모델이 회전키(A, D)를 입력 받았을 때 몸통과 렌즈가 서로 반대 방향으로 회전하는 문제가 있었다. 이는 atan2함수는 반시계 방향을 양의 각도로 계산하는데 OpenGL의 공간에서는 기본적으로 -Z축을 정면으로 바라보도록 설계되었기 때문이었다. 따라서 atan2값에 '-'부호를 추가하여 문제를 해결할 수 있었다.

5-2. 자동 주행 시 역주행 문제

A-star 알고리즘 최초 구현 후 주행을 시작하니까 카메라가 시작지점에서 경로를 따라가는 것이 아니라 벽을 뚫고 목표 지점으로 바로 이동하는 문제가 발생했다. 이유를 분석해보니 이는 구현한 A-star 알고리즘이 목표점에서 부모노드를 타고 내려가서 경로를 저장하므로 결과 리스트에는 목표지점에서 시작지점 순서로 데이터가 쌓였기 때문이다. 이를 해결하기 위해 쌓인 경로를 역순으로 재정렬하는 로직을 추가하였다.

5-3. 자동 주행 시 화면 떨림 및 부자연스러운 회전

처음에는 자동 주행 중에 방향을 회전할 때 순간적으로 팍 튀거나 또는 너무 느리게 회전하는 등 이상한 느낌을 주었다. 이를 위해 현재처럼 보간을 적용하여 부드러운 이동을 가능하게 했다.

5-4. 경로 시각화의 불연속성 문제

최초에 구현한 경로 시각화 코드는 탐색된 경로 위에 그냥 작은 큐브하나를 올려놓았다. 그렇게 하니 경로가 경로 같아 보이지 않고 뚝뚝 끊겨보여 별로라는 느낌이 강하게 들었다. 이는 당연히 노드와 노드 사이의 빈 공간을 고려하지 않고 계산했기 때문에 발생할 수밖에 없었던 문제다. 해결하기 위해 현재 노드와 다음 노드의 평균을 계산하고(중간 지점) 두 노드의 방향벡터를 구해 해당 축으로 큐브의 크기를 1.0f로 늘려 빈공간을 채웠다. 이를 통해 끊어진 점들이 아닌 하나의 이어진 파이프 형태로 보였지만 경로가 꺾이는 부분에 약간의 빈틈이 생겨 얻어진 결과가 그렇게 만족스럽지 않았다. 매우 간단하게 경로의 노드에 cube로 살짝 덮어주니 쉽게 해결할 수 있었다.