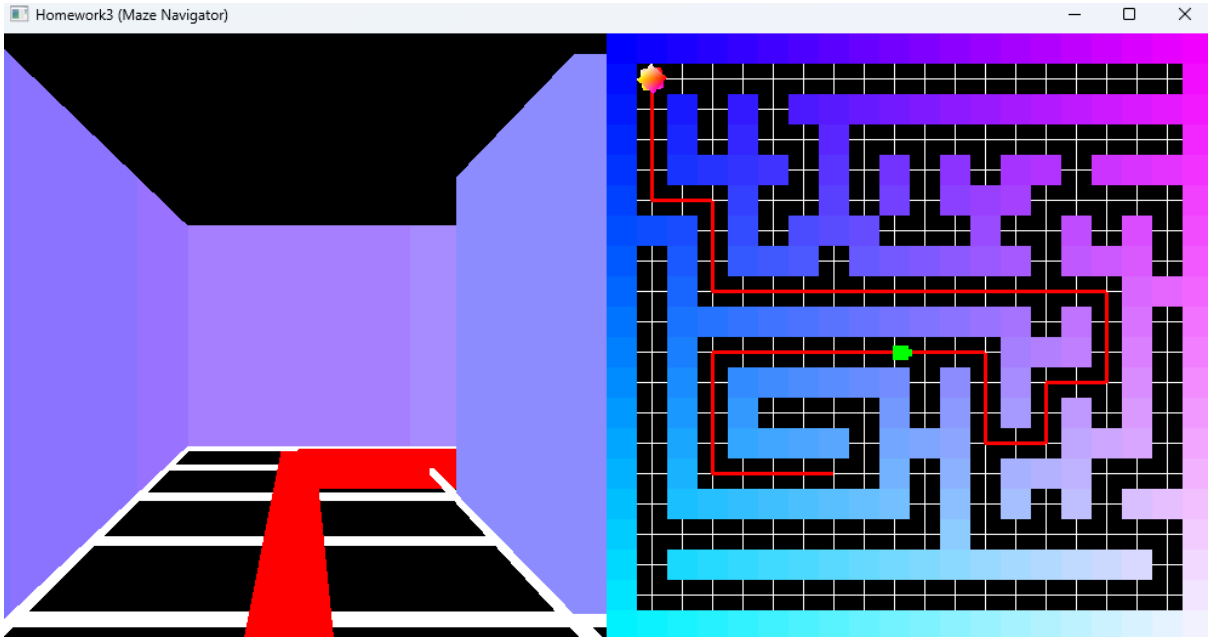
컴퓨터그래픽스 과제 3

소프트웨어학과 20003325 이태혁

실행결과



풀이과정

주어진 코드 이해

각 기능들을 구현하기 전에 먼저 코드에 대한 이해가 필요했다.

실행결과는 두개의 영역으로 분리된다. 왼쪽영역은 view transform 과정에서 focus 벡터를 카메라좌표에 현재 바라보고 있는 방향의 단위벡터(viewDirection)를 더해주어서 viewDirection에 따라서 카메라가 바라보는 방향이 정해지도록 하였고, 오른쪽 영역은 카메라를 y축으로 5만큼 띄운 후에 focus를 원점을 향하게 하여 미로 전체를 내려다보도록 하였다.
왼쪽화면은Perspective Projection 이 적용되어 원근감이 느껴지도록 하고, 오른쪽 화면은 단순히 Orthographic Projection만 적용되었다.

txt파일에서 *이 있는 부분에 cube를 하나씩 그려서 벽을 만든다. 바닥에 grid는 흰색 cube를 매우 얇고 길게 만들어서 그린다.

1. 카메라 회전

카메라는 앞, 뒤로만 움직일 수 있다. a, d 키를 누르면 왼쪽, 오른쪽으로 이동하는 것이 아니라, 카메라가 각각 반시계, 시계방향으로 회전한다.

카메라의 회전을 구현하는데 필요한 작업은 카메라가 처다보는 방향을 회전시키는 것과 오른쪽 화면의 카메라 도형을 회전시키는 것이다.

카메라 도형의 회전

```
mat4 ModelMat = Translate(cameraPos) * RotateY(theta) * Scale(vec3(cameraSi
ze)); // 카메라 몸통

ModelMat = Translate(cameraPos + viewDirection * cameraSize / 2) * RotateY
(theta) * Scale(vec3(cameraSize / 2)); // 카메라 렌즈
```

카메라가 회전하는 각도 theta를 정의했다.

카메라 도형 자체의 회전은 CameraDraw 함수에서 카메라의 몸통과 렌즈를 theta만큼 회전시키면 해결된다. 이 때, rotation을 먼저 한 후에 translate를 해준다.

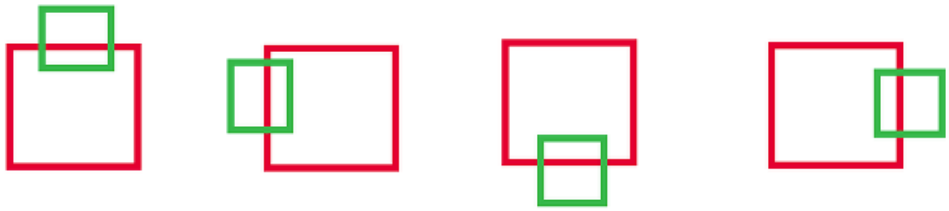
카메라가 바라보는 방향 회전

```
if ((GetAsyncKeyState('A') & 0x8000) == 0x8000) {
    theta += 3.0f;
    float rad = 3.141592 / 180 * theta;
    viewDirection = vec3(-sin(rad), 0, -cos(rad));
  }
```

view transform 과정에서 카메라가 바라보는 focus 지점이 카메라를 중심으로 원형으로 돌아가도록 하는 것이 핵심이다. ‘A’ 를 눌렀을 때 theta는 증가하고, 초기상태인 카메라가 -z 방향 (화면 안으로 들어가는 방향)을 가리키고 있을 때 카메라가 반시계방향으로 회전하면x는 0 → -1 까지 감소하고 z는 -1 → 1까지 증가한다. 따라서 삼각함수의 그래프를 고려해 보았을 때 viewDirection벡터에 x = -sin, z = -cos 을 적용하면 카메라가 회전함에 따라 올바른 정면을 바라보게 된다. ‘D’를 눌렀을 때는 각도만 -theta로 적용해주어서 반대방향으로 회전하게 한다.

2. 벽 충돌 처리

처음에는 벽의 큐브 하나하나와 카메라까지의 거리를 계산하고, 거리가 (카메라 길이 / 2) + (큐브 길이 / 2) 보다 작으면 충돌한 것으로 처리하려고 했다. 하지만 그렇게 하면 카메라와 큐브가 대각선으로 만나는 경우 카메라 일부가 큐브 안으로 들어간 상태에서 충돌처리가 발생한다. 그래서 다른 방법을 생각했다.



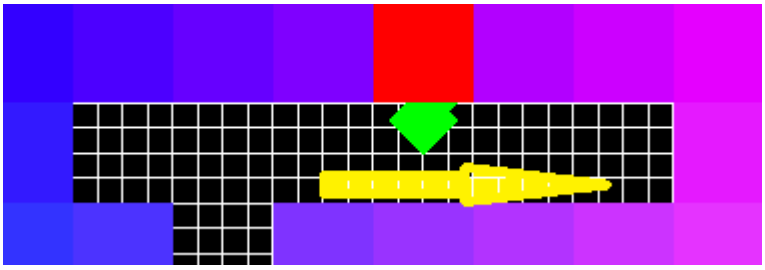
벽의 큐브 하나와 카메라가 충돌하는 경우는 위 그림과 같이 정의된다. 카메라 사각형이 큐브 안으로 들어가는 순간 충돌이 발생한다. 이를 조건문으로 아래 코드처럼 처리했다.

```
if (cameraPos.x + 0.25 >= wallPos.x - 0.5 && cameraPos.x - 0.25 <= wallPos.x + 0.5 && cameraPos.z + 0.25 >= wallPos.z - 0.5 && cameraPos.z - 0.25 <= wallPos.z + 0.5) // 충돌체크
```

wallPos는 큐브의 좌표이며, 큐브 한 변의 길이가 1이므로 각 모서리는 중심에서 0.5만큼 떨어지고, 카메라 한 변의 길이는 0.5이므로, 각 모서리는 카메라 중심에서 0.25만큼 떨어진다.

이렇게 충돌 체크를 하고난 후 충돌 이후의 카메라 위치를 처리해주어야 하는데, 벽에 충돌한 후 카메라가 벽의 반대방향으로 살짝 밀려나게 처리를 해주어야 대각선으로 벽에 부딪힐 때 카메라가 벽을 타고 부드럽게 움직이게 된다.

- 카메라가 벽에 계속 부딪히며 오른쪽으로 벽을 타고 이동함



이 때, 벽에 충돌하는 방향에 따라서 카메라가 밀려나는 방향이 달라지기 때문에 카메라가 벽에 충돌하는 방향을 4가지로 나누어서 따로 처리해주어야 한다.

```
// 충돌체크
    if (cameraPos.x + 0.25 >= wallPos.x - 0.5 && cameraPos.x - 0.25 <= wallPos.
x + 0.5 && cameraPos.z + 0.25 >= wallPos.z - 0.5 && cameraPos.z - 0.25 <= wallP
os.z + 0.5) {

  // 벽의 왼쪽에서 부딪히면
  if (abs((cameraPos.x + 0.25) - (wallPos.x - 0.5)) < 0.1)  cameraPos.x -= 0.1
f;
  // 벽의 오른쪽에서 부딪히면
  if (abs((cameraPos.x - 0.25) - (wallPos.x + 0.5)) < 0.1)  cameraPos.x += 0.1
f;
  // 벽의 아래쪽에서 부딪히면
  if (abs((cameraPos.z - 0.25) - (wallPos.z + 0.5)) < 0.1)  cameraPos.z += 0.1
f;
  // 벽의 위쪽에서 부딪히면
  if (abs((cameraPos.z + 0.25) - (wallPos.z - 0.5)) < 0.1)  cameraPos.z -= 0.1
f;
  }
```

카메라의 한쪽 모서리가 큐브의 한쪽 모서리와 닿으면 벽과 부딪혔다고 판단한다.

부딪히는 방향에 따라 각각 벽의 반대방향으로 cameraPos를 0.1만큼 이동시켜준다.

이제 벽에 부딪히면 부딪힌 벽이 빨간색으로 바뀌도록 작업해야 한다. DrawMaze 함수 내에서 충돌조건을 똑같이 넣어주어 만약 큐브와 카메라가 충돌한다면 해당 큐브의 color는 red가 되도록 해주었다.

시행착오

1. 부동소수 오차

벽에 부딪히는 방향을 정의할 때 카메라의 모서리와 큐브의 모서리가 닿는것을 처음에는 두 모서리의 좌표가 같을 때 라고 코딩했었다.

```
--- 잘못된 코드 ---

// 벽의 왼쪽에서 부딪히면
if (cameraPos.x + 0.25 == wallPos.x - 0.5)  cameraPos.x -= 0.1f;
```

그러나 이렇게 코딩하게 되면 부동소수 오차에 의해 양 변이 완전히 같은 값이 나올 수 없기 때문에 실제로 충돌하더라도 조건문이 동작하지 않는다. 따라서 카메라 모서리와 큐브 모서리 위치의 차가 아주 작은 값 (내 코드의 경우 0.1로 지정하였다) 이하라면 두 위치가 같다고 판단하도록 수정하였다.

2. 실행속도 향상

충돌한 벽이 빨간색으로 바뀌도록 하는 과정에서, 처음에는 충돌체크 여부를 판단하는 collision함수에서 true/false를 리턴하고, DrawMaze 내에서 collision 함수를 호출하여, 만약 충돌한다면 color를 red로 설정하도록 하였다.

```
void DrawMaze()
{
  for (int j = 0; j < MazeSize; j++)
    for (int i = 0; i < MazeSize; i++)
      if (collision() == true)
        vec3 color = vec3(1, 0, 0) ...
```

하지만 이렇게 하면 DrawMaze 안에서 for문 두개가 돌면서 collision함수를 호출하고, collision함수도 충돌 여부를 체크하기 위해 for문 두개가 더 돌아서 총 4중포문이 되어버린다. 따라서 실행속도가 눈에띄게 느려졌다. 결국 DrawMaze 함수 내에서 직접 충돌체크를 하게 하여 2중포문으로 해결하였다.

3. 최단거리 찾기

A* 알고리즘에 대한 이해

출발점에서 목적지까지 가는데 가장 짧은 경로를 찾아내는 알고리즘이다. 길찾기 알고리즘에는 플로이드, 다익스트라, A* 알고리즘이 존재하는데, 플로이드는 맵의 모든 정점에서 다른 모든 정점까지의 경로를 구하는 알고리즘이고, 다익스트라는 특정 출발점에서 다른 모든 정점까지의 경로를 구하는 알고리즘이다. 위의 두 알고리즘은 최적의 경로가 아닌 경로들까지 모두 탐색하기 때문에 길을 찾는 속도가 느리다. 반면 A* 알고리즘은 출발점과 목적지가 주어지면 두 점 사이의 경로만 계산하기 때문에 속도가 상대적으로 빠르다.

A* 알고리즘은 f = g + h 라는 수식을 사용하는데, g는 출발점으로부터 현재 노드까지의 거리이고 h는 현재 노드부터 목적지까지의 거리이다. 그리고 두 값을 더하여 f를 구한다.

열린목록과 닫힌목록이 존재하는데, 열린목록에는 아직 탐색을 하지 않은 노드들을 넣어두고, 닫힌목록에는 탐색을 마친 노드들을 넣는다.

열린목록에 있는 노드중 F가 가장 작은 노드를 선택하고, 해당 노드를 기준으로 이웃노드 8칸을 탐색한다.
이웃노드들중에 장애물이나 이미 탐색을 마친 노드를 만나면(닫힌목록에 있는 노드이면) 무시한다. 그렇지 않는 노드들은 열린목록에 추가한다. 이 때 이웃노드의 부모노드를 현재노드로 저장해둔다.

만약 이웃노드가 이미 열린목록에 있었다면 더 가까운 경로를 찾기 위해서 g값을 비교하여 현재 노드를 거쳐서 가는것이 더 나은 선택인지 판단하고, 만약 현재 노드를 거쳤을 때 g값이 더 작다면, 이웃노드의 ghf값들을 갱신한다.

이렇게 계속 반복하다가 목적지 노드가 열린목록에 추가되면 작업을 종료한다.
그 후에 목적지 노드를 시작으로 부모노드들을 따라 올라가면 그 경로가 바로 최단거리가 된다.

구현 과정

Node 구조체를 만들어서 h, f, g값과 노드의 인덱스값인 (x, z)를 멤버변수로 생성했다. 또한 부모노드의 좌표는 parent 변수에 저장한다.

```
typedef struct Pos {
  int x, z;
}pos;

typedef struct Node {
  int x, z;
  int h = 99, f = 99, g = 99;
  pos parent;
}node;
// F = g + h
```

Q 키를 누르면 findpath 함수가 호출된다. 열린목록과 닫힌목록은 링크드리스트로 구현했다. g, h, f를 계산할 때 현재 카메라 좌표를 기준으로 계산하면 실수값들의 계산이 되므로 복잡해진다. 따라서 화면의 왼쪽 위를 [0, 0]이라고 가정하고, 현재 카메라 좌표를 인덱스로 변환해서 각 노드마다 정수값인 인덱스를 기준으로 계산하였다.

```
  // ----------- 카메라 좌표로부터 start 노드 인덱스 계산 ----------
  node start;
  for (int j = 0; j < MazeSize; j++)
    for (int i = 0; i < MazeSize; i++)
      if (cameraPos.x > -MazeSize / 2.0 + i && cameraPos.x <= -MazeSize /
 2.0 + 1 + i && cameraPos.z > -MazeSize / 2.0 + j && cameraPos.z <= -MazeSi
ze / 2.0 + 1 + j) {
        start.x = i;
        start.z = j;
      }
```

열린목록을 순회하며 가장 작은 f값을 가지고 있는 노드를 선택하고, cur에 저장한다.

```
int minF = openSet.front().f;
    for (node n : openSet)
      if (n.f <= minF) {
        minF = n.f;
```

```
        cur = n;
      }
```

cur의 이웃노드가 벽을 만나는지, 닫힌목록에 있는지 확인하고 만약 열린목록에 없는 노드이면 열린목록에 추가해준다. 이 때, 부모노드의 좌표는 cur의 좌표로 저장한다.

```
// 탐색중인 노드가 열린목록에 없다면

        neighbor.parent.x = cur.x; // 부모노드 설정
        neighbor.parent.z = cur.z; // 부모노드 설정
        openSet.push_back(neighbor); // 이 노드를 열린목록에 추가
```

목적노드가 열린목록에 들어오면 탐색을 종료하고 storepath 함수를 호출한다. 이 함수는 부모노드를 거슬러 올라가며 결과리스트에 시작점부터 도착점까지의 좌표들 모두 저장하는 함수이다.

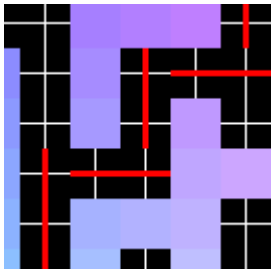
```
for (node n : openSet)
      if (n.x == goal.x && n.z == goal.z) { // openSet에 목적 노드가 들어오면
        storepath(start, n, closedSet);
        return;
      }
```

부모노드를 포인터로 가리키지 않다보니, 부모노드를 찾아내면서 시작점까지 올라가는 방법을 생각하는 것이 힘들었다. 부모노드들은 이미 한번씩은 탐색이 된 노드들이니까 닫힌목록에 모두 존재할 것이고, 닫힌목록을 순회하면서 현재 노드의 부모노드와 좌표가 같은 노드를 찾는 방법으로 구현하였다.

```
while(1){
    for (node i : closedSet) {
      if (tmp.x == i.x && tmp.z == i.z) { // closedSet에서 부모노드를 찾으면
        tmp.x = i.x;
        tmp.z = i.z;
        path.push_back(tmp);
        if (tmp.x == start.x && tmp.z == start.z) {
          drawPath();
          return; // 시작점까지 리스트에 넣었으면 종료
        }
        tmp = i.parent;
        break;
      }
    }
```

경로 좌표들을 모두 찾아내면 이제 화면에 경로를 그려야한다.
경로는 큐브의 Scale을 얇게 조정하고, 길이를 한 칸의 길이와 일치하도록 하는데, 이 때 선을 가로로 그을지 세로로 그을지는 카메라의 이동방향에 따라서 달라진다. 카메라의 이동방향은 현재노드와 이전노드를 비교하여 이전노드와 x값은 같은데 z값만 다르다면 위아래로 움직이는 것과 같이 현재노드와 이전노드를 비교하는 방법으로 구현했다.

또한, 직선이 큐브 안에서 그려지게 되면 그림처럼 엇갈리는 부분이 생긴다.



따라서 카메라가 움직이는 방향마다 직선의 위치를 알맞게 조정해주어 직선이 끊어지지 않도록 하였다.

```
if (tmp.x == i.x) {
      if (tmp.z > i.z) // 위에서 아래로 이동
        ModelMat = Translate(0, -0.5, -0.5) * Translate(getPositionFromInde
x(tmp.x, tmp.z)) * Scale(0.13, 0.13, 1);
      if (tmp.z < i.z) // 아래에서 위로 이동
        ModelMat = Translate(0, -0.5, 0.5) * Translate(getPositionFromIndex
(tmp.x, tmp.z)) * Scale(0.13, 0.13, 1);
```

```
    }
    if (tmp.z == i.z) {
      if (tmp.x < i.x) // 왼쪽에서 오른쪽으로 이동
        ModelMat = Translate(0.5, -0.5, 0) * Translate(getPositionFromIndex
(tmp.x, tmp.z)) * Scale(1, 0.13, 0.13);
      if (tmp.x > i.x) // 오른쪽에서 왼쪽으로 이동
        ModelMat = Translate(-0.5, -0.5, 0) * Translate(getPositionFromInde
x(tmp.x, tmp.z)) * Scale(1, 0.13, 0.13);
    }
```

시행착오

```
-- 잘못된 parnt 코드 ---

typedef struct Node {
  int x, z; // 인덱스 저장
  int h = 99, f = 99, g = 99;
  struct Node* parent;
}node;
```

처음에는 구조체에 Node* parent라는 포인터 변수를 만들어서 부모노드를 가리키게 했었다. 하지만 이렇게 하면 부모노드가 열린목록에서 닫힌목록으로 넘어가는 과정에서 부모노드의 주소가 바뀌고, 경로를 모두 찾은 뒤에 나중에 부모노드를 따라 올라갈 때 엉뚱한 주소에 방문하게 된다. 따라서 포인터를 사용하지 않고, 아예 부모노드의 좌표 자체를 저장하는 방법으로 해결했다.

4. 길 따라가기

스페이스바를 누르면 카메라가 목적지까지 최단거리를 따라간다. 길찾기 단계에서 찾아놓은 좌표들을 순회하여 cameraPos를 해당 좌표들로 순차적으로 이동시킨다.

이 때, 좌표들을 순회하는 반복문을 그냥 돌리게 되면 카메라가 시작점에서 도착점까지 이동하는 과정이 보이지 않고 한번에 순간이동 해 버린다. 따라서 한칸 이동할 때 마다 cameraPos를 조금씩 이동시키면서 Sleep함수를 호출하여 시간을 지연시켜 자연스럽게 움직일 수 있도록 하였다.

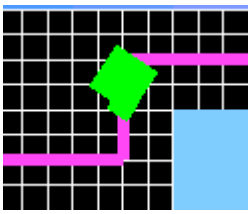
```
if (tmp.x == i.x) {
            if (tmp.z > i.z) {  // 아래에서 위로 이동
              viewDirection = vec3(0, 0, -1);
              theta = 0;
              for (int i = 0; i < 10; i++) {
                cameraPos += vec3(0, 0, -0.1);
                display();
                Sleep(10);
              }
```

이 때도 이동 방향을 찾아내어서 cameraPos를 x방향, z방향의 +, -방향을 고려하며 움직여야 한다. 또한, pre_turn, next_turn 이라는 변수를 만들어서 이동방향이 바뀌는 시점을 판단하여 그 시점에 카메라의 방향이 돌아가도록 하였다.

```
// ------------- 이동 방향이 바뀌면 카메라 시점 변환 ------------

            if (pre_turn == 1 && next_turn == 3) // 우
                viewDirection = vec3(1, 0, 0);

            if (pre_turn == 1 && next_turn == 4) // 좌
              viewDirection = vec3(-1, 0, 0);

            if (pre_turn == 2 && next_turn == 3)  // 좌
              for (int i = 0; i < 10; i++)
                viewDirection = vec3(-1, 0, 0);

            if (pre_turn == 2 && next_turn == 4)  // 우
              viewDirection = vec3(1, 0, 0);

            if (pre_turn == 3 && next_turn == 1) // 상
              viewDirection = vec3(0, 0, -1);


            if (pre_turn == 3 && next_turn == 2) // 하
              viewDirection = vec3(0, 0, 1);

            if (pre_turn == 4 && next_turn == 1)  // 상
              viewDirection = vec3(0, 0, -1);
```

```
            if (pre_turn == 4 && next_turn == 2) // 하
              viewDirection = vec3(0, 0, 1);
```

시행착오



방향이 틀어진 상태로 스페이스바를 누르게 되면 그림처럼 카메라의 방향이 뒤틀린 채 움직이게 된다. 이는 카메라의 방향은 a, d를 누를때 조작되는 theta값에 영향을 받는데, 스페이스바를 누르는 순간 카메라가 처다보는 방향에 따라 theta값을 초기화 해주지 않아서 생긴느 현상이다.

```
if (tmp.z < i.z) { // 위에서 아래로 이동
              viewDirection = vec3(0, 0, 1);
              theta = 180;
```

이런식으로 방향에 따라 theta를 초기화해주어 해결했다.