가상세계

Planning Paths

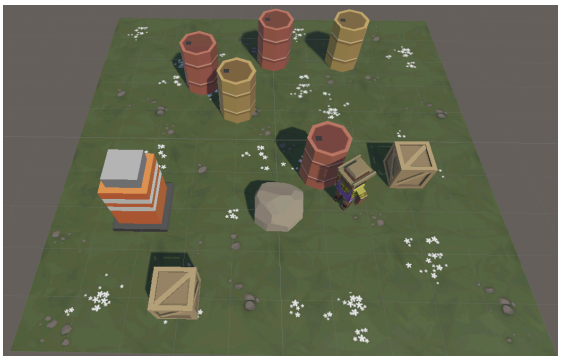
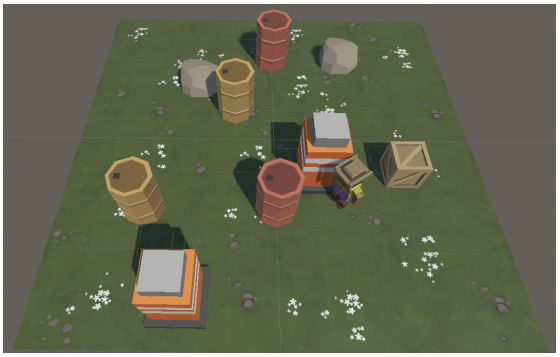
과제 보고서

2024.6.13

2020203052
소프트웨어학부 권빈

Easy1	Placing Obstacles	O
Easy2	Player Animation	O
Normal	Create a second map	O
Hard1	Creating a grid-type graph	O
Hard2	Look in the direction of movement	O
Expert1	Create a movement path using an alogorithm	O
Expert2	Make the movement smooth	O

1.Placing Obstacles

바위나 박스등 너무 큰 obstacle들의 scale을 조정하여 주어진 범위안에 들어가는 크기도 줄였다

```
for (int i = 0; i < 9; i++)
{
    for (int j = 0; j < gsize; j++)
    {
        if (gridgraph[i, j] == 1)
        {
            // Create obstacle
            randomnum = UnityEngine.Random.Range(0, obstacleprefab.Length);
            Vector3 spawnPos = new Vector3(j - 4, 0, 4 - i);
            Instantiate(obstacleprefab[randomnum], spawnPos, obstacleprefab[randomnum].transform.rotation);
        }
        else if (gridgraph[i, j] == 2)
        {
            // Create character
            Vector3 spawnPos = new Vector3(j - 4, 0, 4 - i);
            character = Instantiate(character, spawnPos, character.transform.rotation);
            characterAnimator = character.GetComponent<Animator>(); // Animator 컴포넌트 가져오기
        }
    }
}
```

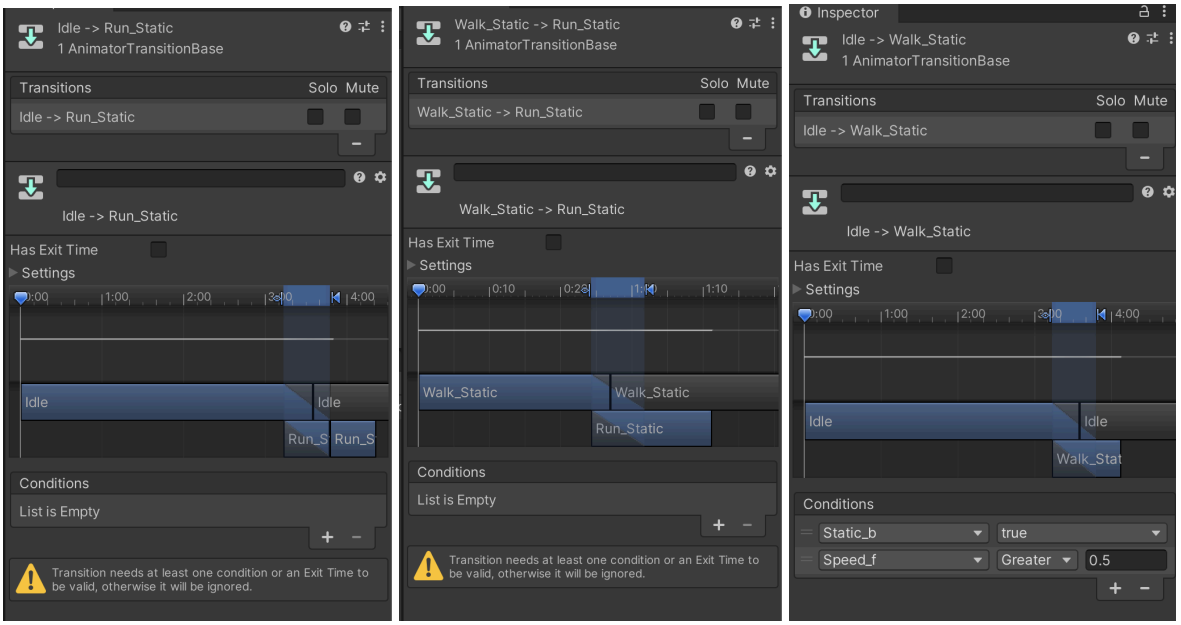
생성은 내가 사용하는 obstacle의 종류 5가지를 배열에 넣고 random함수에서 나오는 숫자번째의 장애물을 그래프를 돌며 장애물이 놓여있어야하는곳 (여기서는 1) 에서 instantiate를 통해 생성하였다
하는김에 캐릭터도 시작위치 (여기서는 2변수에 해당)에 해당하는 칸을 만나면 생성하도록 추가하였다

2.Player Animation

```
if (characterAnimator != null)
{
    characterAnimator.SetFloat("Speed_f", moveSpeed);
}
if (Vector3.Distance(character.transform.position, targetPos) < 0.5f)
{
    pathIndex++;
    if (pathIndex >= path.Count){
    // 목표 지점에 도달한 경우 애니메이션 속도를 0으로 설정
    characterAnimator.SetFloat("Speed_f", 0f);

}
}
```

위에 캐릭터를 생성하는 과정에서 해당 캐릭터의 Animator 컴포넌트를 받아오고 존재하면 지금 설정된 속도를 Speed_f변수에 전달해주고 목표에 도달하면 속도를 0으로 줄이는 코드를 구현한뒤



속도를 바꿀때 run조건을 만족하는 때가 있어서 조건을 삭제시키고 walk에는 현재 속도가 0.5보다 크면 발동하게 하여 걷기와 멈춤을 구현하였다

3.Create a second map

```
static int startx = 1;
참조 3개
static int starty = 5;
참조 2개
static int finishx = 16;
참조 2개
static int finishy = 15;
참조 8개
public static int gsize = 19;
참조 9개
private int[,] gridgraph = new int[gsize, gsize];
참조 3개
```

시작점 끝점, 각 구조물들의 위치와 배열의 크기를 주어진 예시와 같이 바꾸어 기존 알고리즘에 수를 바꾸어 구현하였다

4.Creating a grid-type graph

```
// 1은 장애물, 2는 start, 3은 finish
gridgraph[1, 1] = 2;
gridgraph[1, 4] = 1;
gridgraph[1, 6] = 1;
gridgraph[2, 2] = 1;
gridgraph[3, 3] = 1;
gridgraph[5, 5] = 1;
gridgraph[5, 7] = 1;
gridgraph[6, 1] = 1;
gridgraph[6, 4] = 1;
gridgraph[8, 2] = 1;
gridgraph[6, 5] = 3;
```

```
// 1은 장애물, 2는 start, 3은 finish
gridgraph[startx, starty] = 2;
for (int i = 7; i < 17; i++)
{
    gridgraph[1, i] = 1;
}
for (int i = 3; i < 12; i++)
{
    gridgraph[i, 5] = 1;
}
for (int i = 2; i < 15; i++)
{
    gridgraph[i, 16] = 1;
}
for (int i = 6; i < 17; i++)
{
    gridgraph[14, i] = 1;
}
gridgraph[finishx, finishy] = 3;
```

각 엣지의 가중치가 1인 노드이기 때문에 그냥 2차원 배열로 만들어 사용하였다 이러면 문제점이 이차원 배열의 0,0과 실제 map의 원점이 일치하지 않는다는 점이 있는데

```
Vector3 spawnPos = new Vector3(j - 9, 0, 9 - i);
```

각 위치를 실제로 오차가 나는만큼 보정해주어 문제를 해결하였다

5.Look in the direction of movement

```
Vector3 direction = (targetPos - character.transform.position).normalized;
if (direction != Vector3.zero)
{
    Quaternion lookRotation = Quaternion.LookRotation(direction);
    character.transform.rotation = Quaternion.Slerp(character.transform.rotation, lookRotation, Time.deltaTime * moveSpeed * 4);
}
```

방향을 현재 캐릭터의 위치와 targePos(다음으로 이동해야하는 점)의 차로 벡터를 구하고 unit화 시켜 설정해주었다.
구했던 벡터를 기준으로 LookRotation을 이용해 Quaternion 형식의 돌아야 하는 정보를 입력받은지 rotation y를 lookRotation을 설정해 가는 방향을 바라 보도록 바꾸었다
이때 부드러운 전향을 위해서 Slerp을 사용하였고 이동속도 보다 높은 비율로 rotation을 바꾸게하여 방향을 바꾸기도 전에 새로운 방향으로 바꿔야하는 불상사를 막았다.

6.Create a movement path using an algorithm

알고리즘은 BFS를 선택하였다.
하지만 목표점맞 찾는것이 아니라 그곳까지 가는 과정을 구해야하므로 queue에 넣을때 점의 위치와함께 지금까지 지나온 점들의 배열을 추가하였다.

```
Queue<Tuple<Vector2Int, List<Vector2Int>>> queue = new Queue<Tuple<Vector2Int, List<Vector2Int>>>();
Vector2Int start = new Vector2Int(startx, starty);
queue.Enqueue(Tuple.Create(start, new List<Vector2Int> { start }));
visited[startx, starty] = true;
```

while()문

```
while (queue.Count > 0)
{
    var current = queue.Dequeue();
    Vector2Int currentPos = current.Item1;
    List<Vector2Int> path = current.Item2;

    if (currentPos.x == finishx && currentPos.y == finishy)
    {
        return path;
    }

    for (int i = 0; i < 4; i++)
    {
        int newY = currentPos.x + movey[i];
        int newX = currentPos.y + movex[i];

        if (newY >= 0 && newY < gsize && newX >= 0 && newX < gsize && !visited[newY, newX] && gridgraph[newY, newX] != 1)
        {
            visited[newY, newX] = true;
            List<Vector2Int> newPath = new List<Vector2Int>(path);
            newPath.Add(new Vector2Int(newY, newX));
            queue.Enqueue(Tuple.Create(new Vector2Int(newY, newX), newPath));
        }
    }
}
```

위의 queue의 특별한점 말고는 그냥 일반적인 BFS이다 시작지점부터 상하좌우를 탐색하고 조건(들리지 않았거나 벽이 없거나 범위를 벗어나지 않았거나)
하면 스스로의 path의 자기자신을 넣고 방문표시를 하고 queue에 집어넣어
위치가 finish와 같아질때까지 반복한다.

```
참조 0개
void Update()
{
    if (path != null && pathIndex < path.Count)
    {
        Vector3 targetPos = new Vector3(path[pathIndex].y - 9, 0, 9 - path[pathIndex].x);
        float step = moveSpeed * Time.deltaTime;
        character.transform.position = Vector3.MoveTowards(character.transform.position, targetPos, step);
```

current의 위치가 finish와 같으면 해당 tuple의 경로를 가져와서 way point들을 지나쳐가면서 목표지점까지 가게 만들어주었다.

7.Make the movement smooth

의외로 간단하게 풀린 문제였는데

```
if (Vector3.Distance(character.transform.position, targetPos) < 0.5f)
{
    pathIndex++;
```

캐릭터가 목표 지점에 완전히 가기전에 다음 웨이포인트로 넘어가는 방식으로 직선은 그대로가게되고 웨이포인트가 꺽여있을시 부드럽게 움직이게 된다.
거리가 0.5f인 이유는 여러가지 해봤는데 현재의 속도와 상황에서 가장 부드러운 것 같아서 설정하게 되었다.

Comment

처음에 길을 찾는 알고리즘을 A*로 시도를 했었는데 C#에서 가중치 큐작동 방식을 몰라서 LIST로 만들고 node class의 F(=G+H)값으로 비교하면서
sort하는 방식으로 구현하였는데 생각만큼 잘 안되었다. 제출을 해야하니까 BFS방식으로 다시 코드를 만들고 제출하지만 시간이 날 때 A*로도 구현해보고 싶다.

Youtube 링크

https://youtu.be/8ewqMQHJAF4

https://youtu.be/8ewqMQHJAF4