

가상세계 HW #3. Planning Paths

제출일	2024. 6. 13.(목)	전공	소프트웨어학부
과목	가상세계	학번	2017203082
		이름	김찬진

목차

1. Fulfill table

2. 각 요구사항에 대한 구현방법

3. 과제에 대한 나의 의견

4. 완성본 동영상 링크

1. Fulfil table

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 algorithm	O
Expert2	Make the movement smooth	O (70 %)

2. 각 요구사항에 대한 구현방법

(1) Creating a grid-type graph

- MapGraph, MapGraphTwo, GridManager에 연결됨
- Node 스크립트, Node 프리팹에 연결됨
- 먼저 격자구조의 그래프를 생성하기 위해 3차원 구를 사용하여 Node를 프리팹으로 만들어 주었습니다.
- Node는 자신의 이웃 노드를 담은 neighbors 리스트, 자신의 위치에 장애물이 있음을 알리는 isObstacle, 자신의 번호를 나타내는 nodeNum을 멤버로 가집니다.

```
public class Node : MonoBehaviour
{
    public List<Node> neighbors = new List<Node>();
    public bool isObstacle = false;
    public int nodeNum;
    참조 16개
    public void AddNeighbor(Node neighbor)
    {
        if (!neighbors.Contains(neighbor))
        {
            neighbors.Add(neighbor);
        }
    }
}
```

- 또한 이웃노드가 중복으로 추가되는 것을 막기 위해, AddNeighbor 멤버함수를 사용하였습니다.
- 다음으로, MapGraph, MapGraphTwo 스크립트에서, GenerateGrid() 함수를 사용하여 격자구조로 이루어진 그래프를 생성하였습니다. 과제에서 주어진 대로, firstMap은 9x9, SecondMap은 19x19의 그래프를 생성하였습니다.

- 아래는 **GenerateGrid** 함수의 주요한 내용입니다.

```
nodes = new Node[gridSize, gridSize];
int nodeNum = 0;
for (int x = 0; x < gridSize; x++)
{
    for (int y = 0; y < gridSize; y++)
    {
        Vector3 position = new Vector3(x * spacing, -1, y * spacing);
        GameObject nodeObject = Instantiate(nodePrefab, position, Quaternion.identity, transform);
        Node node = nodeObject.GetComponent<Node>();
        nodes[x, y] = node;
        node.nodeNum = nodeNum;

        Debug.Log($"Node created at index ({x},{y}) with nodeNum: {nodeNum}");

        if (ObstacleSpawn(position, nodeNum))
        {
            node.isObstacle = true;
        }
        nodeNum++;
    }
}
```

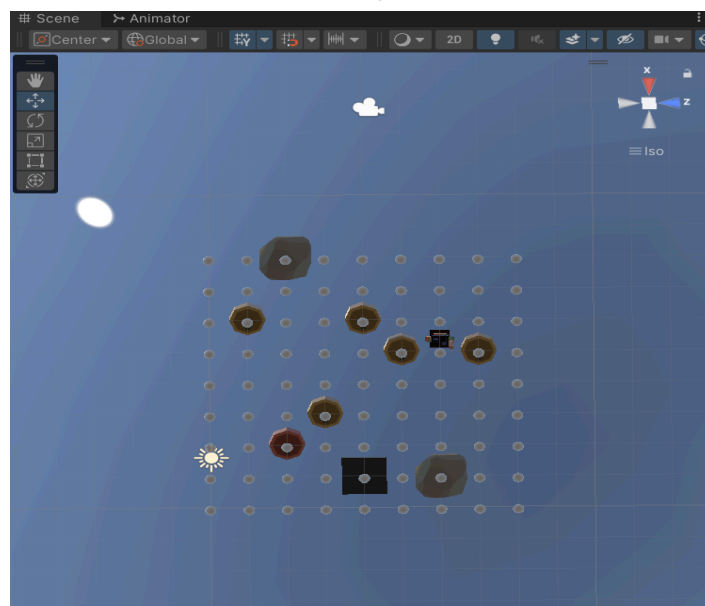
- **gridSize**는 firstMap은 9, SecondMap은 19입니다.
- **spacing**은 1 이며, 격자그래프 생성을 위해, **nodes** 2차원 배열을 만들고, 이를 순회하며 노드 오브젝트를 생성하였습니다. 이때, 생성된 노드 오브젝트가 맵에서 보이지 않게 하기 위해 **y**값을 -1로 설정하였습니다.
- 여기서 매번 생성된 오브젝트에서 **Node** 컴포넌트를 가져와 **nodes** 배열에 저장하고, 노드 번호를 설정합니다.
- **ObstacleSpawn** 함수는 뒤에 장애물 생성 부분에 설명하겠습니다.
- 다음으로 그래프 엣지 연결입니다. 인접한 노드에만 엣지를 연결하기 위해, 한번 **nodes** 를 순회하면서 매번 현재 노드 기준, 상하좌우 노드를 이웃으로 설정하였습니다.

```

for (int x = 0; x < gridSize; x++)
{
    for (int y = 0; y < gridSize; y++)
    {
        Node node = nodes[x, y];
        if (x > 0)
        {
            Node leftNeighbor = nodes[x - 1, y];
            node.AddNeighbor(leftNeighbor);
            leftNeighbor.AddNeighbor(node);
        }
        if (y > 0)
        {
            Node bottomNeighbor = nodes[x, y - 1];
            node.AddNeighbor(bottomNeighbor);
            bottomNeighbor.AddNeighbor(node);
        }
        if (x < gridSize - 1)
        {
            Node rightNeighbor = nodes[x + 1, y];
            node.AddNeighbor(rightNeighbor);
            rightNeighbor.AddNeighbor(node);
        }
        if (y < gridSize - 1)
        {
            Node topNeighbor = nodes[x, y + 1];
            node.AddNeighbor(topNeighbor);
            topNeighbor.AddNeighbor(node);
        }
    }
}

```

- $x > 0$: 현재 노드를 왼쪽 노드와 연결합니다
- $y > 0$: 현재 노드를 아래쪽 노드와 연결합니다
- $x < \text{gridSize} - 1$: 현재 노드의 오른쪽 노드와 연결합니다.
- $y < \text{gridSize} - 1$: 현재 노드의 위쪽 노드와 연결합니다.
- 결과적으로 **nodes** 배열에 격자그래프가 생성되며, 아래는 그래프가 생성된 사진입니다.(맵 아래쪽에서 캡처하였습니다)



- 격자 모양의 하얀색 구가 노드입니다.

(2) Placing Obstacles

- MapGraph, MapGraphTwo, GridManager에 연결됨
- Node 스크립트, Node 프리팹에 연결됨
- 과제에서 제시된 위치에 장애물을 설치하기 위해, MapGraph스크립트의 ObstacleSpawn함수를 사용하였습니다.
- ObstacleSpawn함수에서는 switch case 문으로 설정된 노드의 번호에 장애물을 설치하며, true를 리턴합니다.

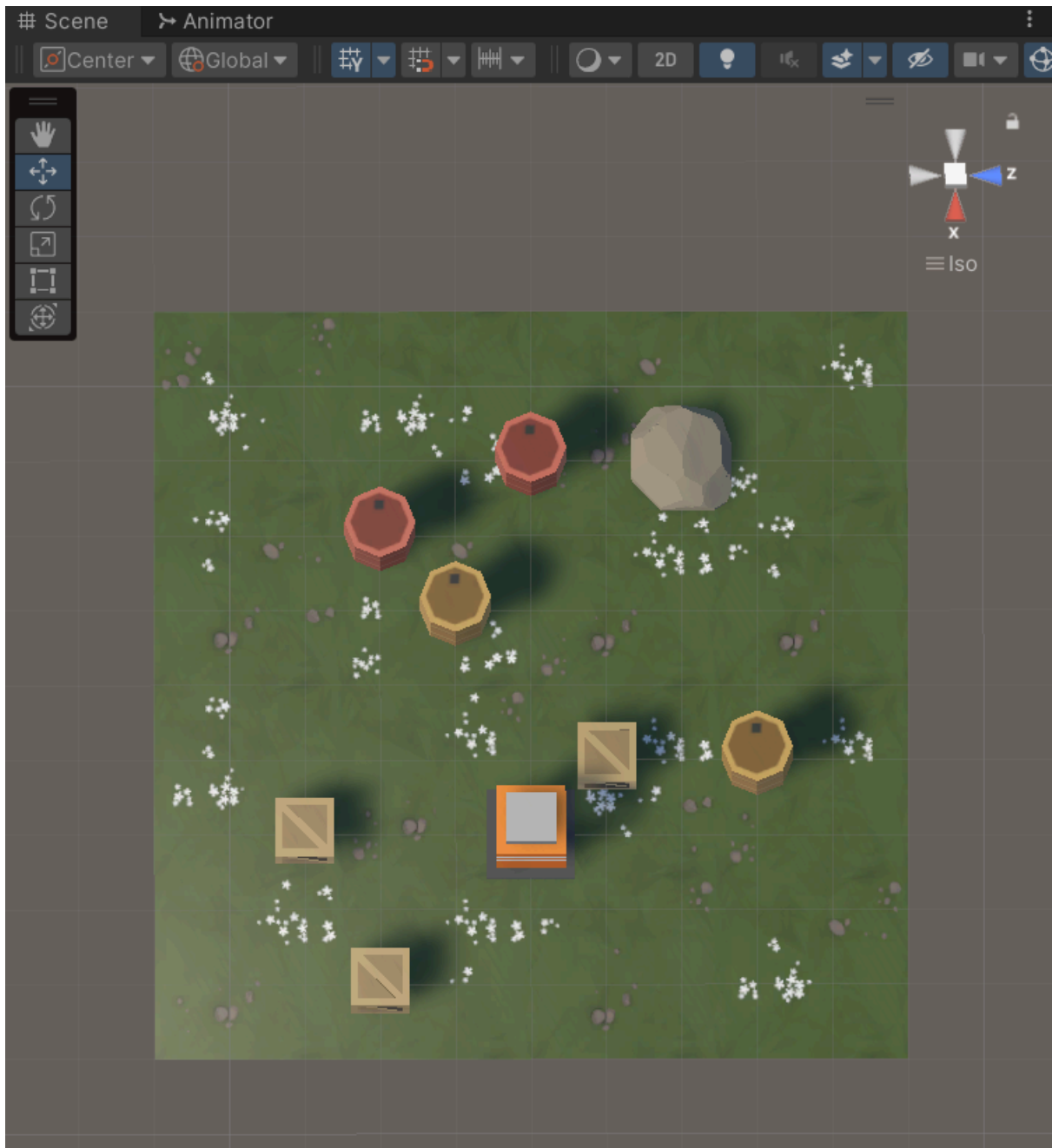
```
bool ObstacleSpawn(Vector3 position,int number)
{
    switch (number)
    {
        case 13:
        case 15:
        case 20:
        case 30:
        case 50:
        case 52:
        case 55:
        case 58:
        case 74:
            position.y = 0;
            Instantiate(obstaclePrefabs[Random.Range(0, 5)], position, Quaternion.identity, transform);
            return true;
    }
    return false;
}
```

- 이후 리턴된 true는 GenerateGrid 함수내부에서 장애물이 위치한 곳에 있는 노드의 isObstacle 변수를 true로 바꾸어줍니다.

```
if (ObstacleSpawn(position, nodeNum))
{
    node.isObstacle = true;
}
```

- 이를 통해, 장애물을 생성하고 이후 최단경로를 구할 때, 장애물이 있는 노드를 제외한 경로를 탐색합니다.
- 또한 장애물은 미리 프리팹화 되어있는 4개의 장애물 중 하나로 무작위로 생성됩니다.

- 아래는 장애물이 설치된 **firstMap**의 사진입니다.



(3) Create a movement path using an algorithm

- MapGraph, MapGraphTwo, GridManager에 연결됨
- 그래프의 모든 엣지가 1이므로 Bfs를 사용하여, 시작 노드와 최종 노드 간의 최단거리를 구하였습니다.
- FindPath 함수에서, 시작 노드번호와 도착 노드번호를 인자로 받은 후, BFS함수와 MakePath 함수를 사용하여 최단경로를 생성하였습니다.
- 이후, 생성된 최단경로를 FindPath가 받아 디버그 로그로, 경로를 출력합니다.

```
void FindPath(int startNodeIndex, int endNodeIndex)
{
    int startX = startNodeIndex / gridSize;
    int startY = startNodeIndex % gridSize;
    int endX = endNodeIndex / gridSize;
    int endY = endNodeIndex % gridSize;

    Node startNode = nodes[startX, startY];
    Node endNode = nodes[endX, endY];
}
```

- FindPath 함수에서, startNodeIndex와 endNodeIndex를 사용하여 시작노드와 도착노드의 그리드 좌표를 계산합니다.
- 그래프를 생성할 때, 배열의 열부터 노드번호가 증가하였으므로, startX,endX는 노드의 행, startY, endY는 노드의 열 인덱스 입니다.
- 그 다음 ,계산된 좌표로 nodes 배열에서 시작노드와 도착노드를 가져옵니다.
- 이후, path = BFS(startNode, endNode); 를 통해 Bfs를 실행합니다.(최단경로 path에 저장)

- 다음으로, BFS 함수입니다.

```
List<Node> BFS(Node start, Node goal)
{
    Queue<Node> frontq = new Queue<Node>();
    frontq.Enqueue(start);

    Dictionary<Node, Node> cameFrom = new Dictionary<Node, Node>();
    cameFrom[start] = null;

    while (frontq.Count > 0)
    {
        Node current = frontq.Dequeue();

        if (current == goal)
        {
            return MakePath(cameFrom, current);
        }

        foreach (Node neighbor in current.neighbors)
        {
            if (!neighbor.isObstacle && !cameFrom.ContainsKey(neighbor))
            {
                frontq.Enqueue(neighbor);
                cameFrom[neighbor] = current;
            }
        }
    }

    return null;
}
```

- 너비우선탐색에 사용할 큐를 생성하고, 시작 노드를 큐에 추가합니다
- 각 노드의 출처(어디에서 왔는지)를 저장할 딕셔너리 **cameFrom**을 생성하고, 시작 노드를 **null**로 설정합니다 (탐색의 출발노드이므로, **null**로 설정하였습니다.)
- 이후, 아래 **while** 문에서는 **frontq**에서 현재노드를 가져와, 현재 노드가 도착노드(**goal**)인 경우, **MakePath**함수로 최단경로를 구성하여 리턴합니다.
- 도착노드가 아닌 경우, **foreach** 문을 통해 이웃 노드가 장애물이 아니고, 아직 방문하지 않은 경우 **frontq**에 추가하고 **cameFrom**에 현재 노드(**current**)에서 왔음을 기록합니다.

- 다음은 MakePath 함수입니다.

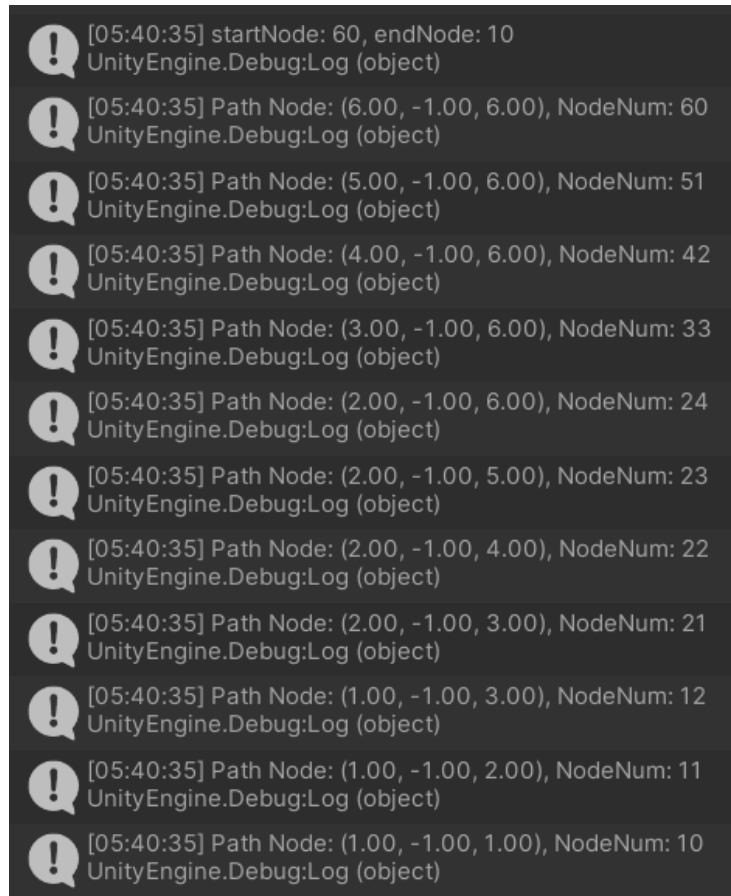
```
List<Node> MakePath(Dictionary<Node, Node> cameFrom, Node current)
{
    List<Node> totalPath = new List<Node> { current };

    while (cameFrom[current] != null)
    {
        current = cameFrom[current];
        totalPath.Add(current);
    }

    totalPath.Reverse();
    return totalPath;
}
```

- 현재 노드(current)를 시작으로, totalPath(시작노드부터, 도착노드까지의 경로)를 만듭니다.
- while 문을 사용하여, cameFrom 딕셔너리의 노드가 빌때까지, totalPath 에 노드들을 추가합니다.
- 이렇게 만들어진 경로는, 도착 노드부터 시작노드까지의 경로이므로, Reverse 함수로 역순을 뒤집어 리턴합니다.
- 이렇게 만들어진 경로는, 최단 경로입니다.(각 노드간 엣지의 가중치가 모두 1인 조건이 있으므로)

- 이후, **start**함수에서 **FindPath(60,10);**을 호출하여(**FirstMap**인 경우) 최단경로를 생성합니다. 아래는 생성된 경로를 디버그 로그로 출력한 사진입니다.(**SecondMap**에서도 동일하게 동작합니다)
- **FindPath** 함수 아래쪽에 **path** 의 경로를 보여주는 디버그 로그 코드가 있습니다.



```
[05:40:35] startNode: 60, endNode: 10
UnityEngine.Debug:Log (object)

[05:40:35] Path Node: (6.00, -1.00, 6.00), NodeNum: 60
UnityEngine.Debug:Log (object)

[05:40:35] Path Node: (5.00, -1.00, 6.00), NodeNum: 51
UnityEngine.Debug:Log (object)

[05:40:35] Path Node: (4.00, -1.00, 6.00), NodeNum: 42
UnityEngine.Debug:Log (object)

[05:40:35] Path Node: (3.00, -1.00, 6.00), NodeNum: 33
UnityEngine.Debug:Log (object)

[05:40:35] Path Node: (2.00, -1.00, 6.00), NodeNum: 24
UnityEngine.Debug:Log (object)

[05:40:35] Path Node: (2.00, -1.00, 5.00), NodeNum: 23
UnityEngine.Debug:Log (object)

[05:40:35] Path Node: (2.00, -1.00, 4.00), NodeNum: 22
UnityEngine.Debug:Log (object)

[05:40:35] Path Node: (2.00, -1.00, 3.00), NodeNum: 21
UnityEngine.Debug:Log (object)

[05:40:35] Path Node: (1.00, -1.00, 3.00), NodeNum: 12
UnityEngine.Debug:Log (object)

[05:40:35] Path Node: (1.00, -1.00, 2.00), NodeNum: 11
UnityEngine.Debug:Log (object)

[05:40:35] Path Node: (1.00, -1.00, 1.00), NodeNum: 10
UnityEngine.Debug:Log (object)
```

(4) Make the movement smooth

- BezierCurve 스크립트
- PlayerController, PlayerControllerTwo 스크립트, Player에 연결
- 수업내용 5장 애니메이션에서 배운 BezierCurve를 사용하여, 앞서 구한 최단경로를 플레이어가 지나갈 때, 곡선으로 부드럽게 지나가게 하였습니다.
- 먼저, BezierCurve 스크립트입니다.

```
public static class BezierCurve
{
    참조 2개
    public static Vector3 GetPoint(Vector3 p0, Vector3 p1, Vector3 p2, Vector3 p3, float t)
    {
        float u = 1 - t;
        float tt = t * t;
        float uu = u * u;
        float uuu = uu * u;
        float ttt = tt * t;

        Vector3 p = uuu * p0; // (1-t)^3*p0
        p += 3 * uu * t * p1; // 3*t*(1-t)^2*p1
        p += 3 * u * tt * p2; // 3*t^2*(1-t)*p2
        p += ttt * p3;       // t^3*p3

        return p;
    }
}
```

- Bezier곡선을 구하는 식인,
$$p(t) = (1-t)^3p_0 + 3t(1-t)^2p_1 + 3t^2(1-t)p_2 + t^3p_3$$
를 사용하여, t값에 따른 곡선의 좌표를 리턴합니다.

- 다음으로, PlayerController 스크립트입니다.

```
void Start()
{
    playerAnim = GetComponent<Animator>();
    mapgraph = GameObject.Find("GridManager").GetComponent<MapGraph>();
    path = mapgraph.path;

    OnGround();
    GenerateBezierPoints();
}
```

- 먼저 start 함수에서, GridManager로부터 MapGraph 컴포넌트를 가져와, BFS를 통해 생성한 최단경로인 path를 현재 스크립트의 path에 저장합니다.
- 이후, Onground함수, GenerateBezierPoints 함수를 통해 Bezier곡선을 만들어 냅니다.
- 먼저 Onground 함수입니다.

```
void OnGround()
{
    for (int i = 0; i < path.Count; i++)
    {
        groundPath.Add(path[i].transform.position + new Vector3(0, 1, 0));
    }
}
```

- 이전에, 맵을 가리지 않게 하기 위해 노드들의 y값이 -1 인 상태로 저장되었는데, 이를 다시 0으로 되돌립니다.
- 이후 이 노드들을 groundPath 리스트에 저장합니다.

- 다음으로 GenerateBezierPoints 함수입니다.

```
void GenerateBezierPoints()
{
    int groundPathPoint = 0;
    while (groundPathPoint <= groundPath.Count-4)
    {
        Vector3 p0 = groundPath[groundPathPoint];
        Vector3 p1 = groundPath[groundPathPoint + 1];
        Vector3 p2 = groundPath[groundPathPoint + 2];
        Vector3 p3 = groundPath[groundPathPoint + 3];

        for (int i = 0; i < 20; i++)
        {
            float segment = (0.05f * i + 0.05f);
            Vector3 bezierPoint = BezierCurve.GetPoint(p0, p1, p2, p3, segment);
            bezierPoints.Add(bezierPoint);
        }
        groundPathPoint += 3;
    }

    while(groundPathPoint < groundPath.Count - 1)
    {
        bezierPoints.Add(groundPath[groundPathPoint+1]);
        groundPathPoint++;
    }
}
```

- bezier 커브를 생성하기 위해 점 4개를 순차적으로 groundPath로부터 가져와, 곡선을 구성하는 점인 bezierPoint를 20개 만들어 냅니다.(세그먼트 사용) 간격은 0.05f 로 설정하였습니다.
- 인덱스를 초과하는 경우가 생기지 않게 하기 위해, while (groundPathPoint <= groundPath.Count-4)을 통해, 만들 수 있는 만큼의 bezier 곡선을 만들었고(bezierPoints에 점들을 추가함) , 만약 곡선을 생성했음에도 남은 좌표가 groundPath에 있다면, 다음 반복문 while(groundPathPoint < groundPath.Count - 1) 을 통해 마저 bezierPoints에 추가해 주었습니다.

- 마지막으로 **Move** 함수입니다. 이 함수는 매 프레임 호출됩니다.

```
public void Move()
{
    if (currentbezierPoint < bezierPoints.Count)
    {
        Vector3 nextPath = bezierPoints[currentbezierPoint];

        transform.LookAt(nextPath);
        transform.position = Vector3.MoveTowards(transform.position, nextPath, speed * Time.deltaTime);

        if (Vector3.Distance(transform.position, nextPath) < 0.01f)
        {
            currentbezierPoint++;
        }
    }
}
```

- nextPath에 bezierPoints에 있는 좌표를 하나씩 받아, 해당 nextPath 를 따라 플레이어가 이동하도록 하였습니다.
- 이때, 플레이어가 가는 방향으로 부드럽게 회전하게 하기 위해, LookAt 함수를 사용하였습니다.
- 만약, $\text{Vector3.Distance}(\text{transform.position}, \text{nextPath}) < 0.01f$ 이라면 즉, 현재 위치와 nextPath와의 거리가 0.01f 미만이라면(거의 다 왔다면) currentbezierPoint를 1 증가시켜 다음 경로로 이동하게 하였습니다.
- 그러나, 실제로 실행한 결과 간헐적으로 bezierCurve가 생성되지 않아 플레이어가 부드럽게 회전하지 않았습니다.

- 아래는 예시 사진입니다.



- 꺾임이 발생하는 지역을 곡선으로 지나가는 중입니다.

(5) Look in the direction of movement

- PlayerController, PlayerControllerTwo 스크립트, Player에 연결
- 앞서 만들어진 bezierPoints에 있는 점들을 향해 플레이어가 부드럽게 회전하게 하기 위해, LookAt 함수를 사용하였습니다.

```
public void Move()
{
    if (currentbezierPoint < bezierPoints.Count)
    {
        Vector3 nextPath = bezierPoints[currentbezierPoint];

        transform.LookAt(nextPath);
        transform.position = Vector3.MoveTowards(transform.position, nextPath, speed * Time.deltaTime);

        if (Vector3.Distance(transform.position, nextPath) < 0.01f)
        {
            currentbezierPoint++;
        }
    }
}
```

- Move 함수를 통해, 플레이어는 bezierPoints에 있는 점들을 향해 하나씩 따라갑니다. 이때, LookAt 함수를 사용하여 해당 점을 바라보게 해서, 자연스럽게 회전하도록 하였습니다.
- Move함수는 매 프레임 호출됩니다.
- 아래는 예시 사진입니다.



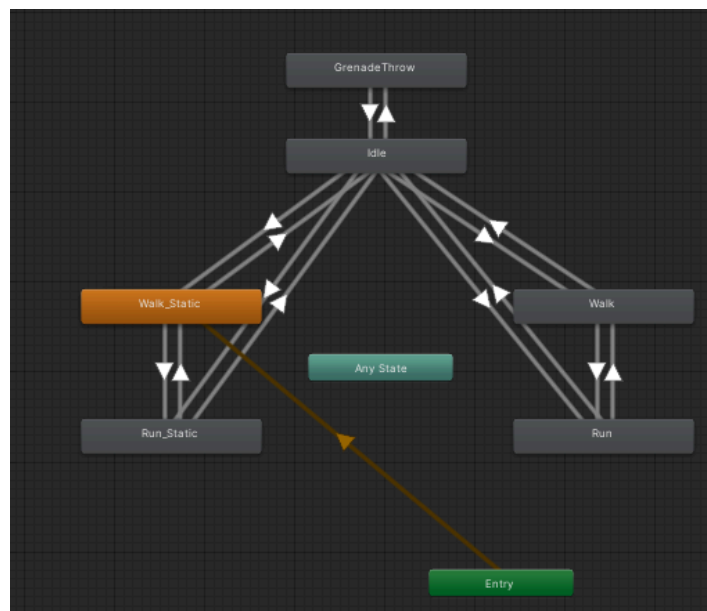
(6) Create a second map

- SecondMap 씬
- MapGraphTwo 스크립트, GridManager에 연결
- PlayerControllerTwo 스크립트, Player에 연결
- FirstMap 과 동일한 오브젝트들을 가진 SecondMap 입니다.
MapGraphTwo에서는 girdSize가 19입니다.
- FirstMap 의 장애물의 배치와 다른 장애물 배치가 필요하기
때문에 앞서 설명한 **ObstacleSpawn** 함수의 **case** 에 과제에서
제시된 장애물의 위치번호를 입력했습니다.
- 최단경로를 찾는 **Bfs** 알고리즘과, 최단경로를 통해 만들어진
곡선 경로를 플레이어가 따라가게 하는 알고리즘은
동일합니다.(단, 시작 노드와 최종 노드의 번호는 다릅니다.)
- 아래는 장애물이 생성된 사진입니다.



(7) Player Animation

- PlayerController, PlayerControllerTwo 스크립트, Player에 연결
- 플레이어에 해당하는 최단 경로를 따라가는 농부가 **0.25f** 이상일 때는 걷는 **Walk_Static** 애니메이션을, 그 이하일 때는 **Idle** 애니메이션을 지정하였습니다.
- 디폴트를 **Walk_Static**으로 지정해 처음 시작할 때는 걷는 모션을 가지고, 최종 목적지에 다다르면 속도를 줄여 **Idle** 이 되도록 하였습니다.



- 디폴트는 **Walk_Static**입니다

```
Move();  
if (Mathf.Abs(transform.position.x - bezierPoints[bezierPoints.Count - 1].x) < 0.01f  
    && Mathf.Abs(transform.position.z - bezierPoints[bezierPoints.Count - 1].z) < 0.01f)  
{  
    playerAnim.SetFloat("Speed_f", 0.0f);  
}
```

- 목표 지점에 다다른 경우(실제로는 목표 지점과 플레이어의 위치의 거리가 **0.01f** 미만인 경우)
SetFloat함수로 **Speed_f**값을 **0.0f** 로 설정해 **Idle** 상태가 되게 하였습니다.

- **start**함수에서는 **GetComponent<Animator>()**를 통해 애니메이터와 연결하였습니다.
- 아래는 **Walk_static** 일때와 **Idle** 상태일 때의 모습입니다.



3. 과제에 대한 나의 의견

(1) 가장 구현하기 힘들었던 부분

- **Bezier Curve** 를 구현하기 위해 네개의 기준점들로부터 생성된 새로운 곡선 점 좌표를 플레이어가 따라가게 하였는데, 간헐적으로 커브가 생겨야할 부분에 커브가 만들어지지 않았습니다. 이 부분을 끝내 해결하지 못해 아쉽습니다.

(2) 향후 이루고 싶은 목표

- 이번학기 가상세계 수업을 수강하면서 알게 된 내용들을 토대로, 향후 프로그래밍 할 때 다양한 부분에서 활용하고 싶습니다. 그래서 높은 수준의 퀄리티를 보이는 사람이 되었으면 합니다.

4. 완성본 동영상 링크

- <https://youtu.be/4SQU49sC3Uc>