BFS与DFS的理论已经了解的差不多了，开始实践

BFS

广度优先算法

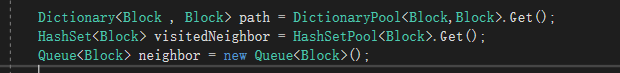
2024年12月30日23点08分

图已构建完毕

图片包含 游戏机, 游戏, 门, 画

AI 生成的内容可能不正确。

算法定义：找自己的邻居A，判断是否为目标，是，结束。不是，再找下个邻居B，同时获取邻居的邻居。



三个容器。

path：

key：邻居的邻居

value：邻居  
手机屏幕截图

AI 生成的内容可能不正确。

以此图为例，黄色的A、B、C作为键，橙色的A作为值。

用于当发现目标时，可以根据目标找到起点。

visitedNeighbor：用于存放已经访问过的邻居，选择hashset的原因为：contains的时间复杂度为o1，且该容器不需要其他任何操作。

neighbor：用于存放邻居、邻居的邻居、邻居的邻居的邻居。。。。。。。。

第一步：构建图（找到邻居）

private void InitMap()

{

//对map中的每个对象都添加自定义的block。block中包含一个容器，用于存放自己的所有邻居

for (int i = 0; i < transform.childCount; i++)

{

Block block = transform.GetChild(i).gameObject.AddComponent<Block>();

\_blockMap.TryAdd(new Vector2Int(Mathf.RoundToInt(block.transform.position.x) , Mathf.RoundToInt(block.transform.position.z)) , block);

}

Block neighbor;

foreach (var item in \_blockMap)

{

if (\_blockMap.TryGetValue(item.Key + Vector2Int.right , out neighbor))

{

item.Value.Neighbors.AddNeighbor(neighbor);

neighbor.Neighbors.AddNeighbor(item.Value);

}

if (\_blockMap.TryGetValue(item.Key + Vector2Int.up, out neighbor))

{

item.Value.Neighbors.AddNeighbor(neighbor);

neighbor.Neighbors.AddNeighbor(item.Value);

}

}

}

第二步：找终点

//初始值，终点

neighbor.Enqueue(\_terminal);

while (neighbor.TryDequeue(out Block block))

{

//交互方式，根据点击的按钮，确定是否要按步执行。用于了解算法的工作原理。

if (!\_goToGoal)

{

while (!\_nextStep)

yield return null;

\_nextStep = false;

}

//将当前邻居标记为已访问过

visitedNeighbor.Add(block);

//开始访问邻居

foreach (var item in block.Neighbors)

{

if (!visitedNeighbor.Contains(item))

{

//保存来时路，用于确定路径。

path[item] = block;

//将邻居的邻居存入访问队列

neighbor.Enqueue(item);

item.gameObject.GetComponent<MeshRenderer>().material.color = Color.cyan;

}

}

if (block == \_begining)

{

Global.MDebug.Log($"path find");

break;

}

}

这里用的是先确定终点再去寻找起点的方式，因为需要player从起点走向终点。

图表, 树状图

AI 生成的内容可能不正确。

2025年1月1日15点22分：自测试通过。

2025年1月4日11点37分

private IEnumerator DFS()

{

//DFS寻路入口

if (\_interaction)

{

yield break;

}

\_interaction = true;

//访问过的对象

HashSet<Block> visitedNeighbor = HashSetPool<Block>.Get();

//需要访问的对象

Stack<Block> blocks = new Stack<Block>();

//路径

Stack<Block> path = new Stack<Block>();

//从终点开始查找，方便

path.Push(\_terminal);

blocks.Push(\_terminal);

visitedNeighbor.Add(\_terminal);

yield return StartCoroutine(DFS\_Tranverse(visitedNeighbor , blocks,path));

while (\_curBlock != \_terminal)

{

yield return StartCoroutine(WaitInteraction());

Global.MDebug.Log($"{\_curBlock.name}");

Block block = path.Pop();

//TODO 移动效果

\_player.transform.position = block.transform.position + Vector3.up;

yield return new WaitForSeconds(0.5f);

block.gameObject.GetComponent<MeshRenderer>().material.color = Color.white;

\_curBlock = block;

}

//移动结束，DFS寻路完成

for (int i = 0; i < transform.childCount; i++)

{

transform.GetChild(i).GetComponent<MeshRenderer>().material.color = Color.black;

}

ResetPath();

\_goToGoal = false;

\_interaction = false;

dfs\_complete = false;

HashSetPool<Block>.Release(visitedNeighbor);

}

private IEnumerator DFS\_Tranverse(HashSet<Block> pIn\_VisitedNeighbor, Stack<Block> pIn\_Blocks , Stack<Block> pIn\_Path)

{

//弹出一个block，并遍历其邻居。（这里存在问题，关于邻居的选择，当前用的是如果A的邻居存在B，那么B的邻居中也必定存在A。）

Block block = pIn\_Blocks.Pop();

foreach (var item in block.Neighbors)

{

yield return new WaitForSeconds(0.2f);

if (item == \_begining)

{

//找到起点，退出所有协程。

dfs\_complete = true;

yield break;

}

if (!pIn\_VisitedNeighbor.Contains(item))

{

//将当前对象标记为访问过

pIn\_VisitedNeighbor.Add(item);

pIn\_Blocks.Push(item);

pIn\_Path.Push(item);

item.gameObject.GetComponent<MeshRenderer>().material.color = Color.cyan;

//一条路走到黑

yield return StartCoroutine(DFS\_Tranverse(pIn\_VisitedNeighbor, pIn\_Blocks, pIn\_Path));

}

if (dfs\_complete)

{

yield break;

}

}

//两种情况可以走到这里。1、当前block不存在邻居。2、当前block的所有邻居都不是起点。

pIn\_Path.Pop().gameObject.GetComponent<MeshRenderer>().material.color = Color.red;

}

电脑屏幕的照片上有文字

AI 生成的内容可能不正确。

自测试通过。

Astat

2025年1月4日12点22分

图片包含 树状图

AI 生成的内容可能不正确。

存在路径断掉的情况

2025年1月4日12点59分

问题已解决目前代码如下：

private IEnumerator AStar()

{

//AS寻路入口

if (\_interaction)

{

yield break;

}

\_interaction = true;

//好像存在路径中断的情况，且将修改全部颜色的函数放置此处

for (int i = 0; i < transform.childCount; i++)

{

transform.GetChild(i).GetComponent<MeshRenderer>().material.color = Color.black;

}

HashSet<Block> visitedNeighbor = HashSetPool<Block>.Get();

//需要访问的对象

Block nextStep;

//路径

Stack<Block> path = new Stack<Block>();

//从终点开始查找，方便

nextStep = \_terminal;

path.Push(\_terminal);

while (nextStep != null)

{

//单路径时死循环

//path.Push(block);

yield return new WaitForSeconds(0.2f);

Block minDisBlock = nextStep.GetMinDisBlockToGoal(\_begining, visitedNeighbor);

if (minDisBlock != null)

{

//路径可能断掉

//path.Push(minDisBlock);

path.Push(nextStep);

nextStep.gameObject.GetComponent<MeshRenderer>().material.color = Color.cyan;

nextStep = minDisBlock;

if (minDisBlock == \_begining)

{

break;

}

}

else

{

if (path.TryPop(out Block temp))

{

nextStep = temp;

temp.gameObject.GetComponent<MeshRenderer>().material.color = Color.red;

}

}

}

while (\_curBlock != \_terminal)

{

yield return StartCoroutine(WaitInteraction());

Global.MDebug.Log($"{\_curBlock.name}");

Block block = path.Pop();

//TODO 移动效果

\_player.transform.position = block.transform.position + Vector3.up;

yield return new WaitForSeconds(0.5f);

block.gameObject.GetComponent<MeshRenderer>().material.color = Color.white;

\_curBlock = block;

}

//移动结束，DFS寻路完成

ResetPath();

\_goToGoal = false;

\_interaction = false;

HashSetPool<Block>.Release(visitedNeighbor);

}

//启发函数，寻找当前欧几里得距离最小的一个邻居

public Block GetMinDisBlockToGoal(Block pIn\_Goal ,HashSet<Block> pIn\_Ignore)

{

float dis = float.MaxValue;

Block \_minDisBlock = null;

foreach (var item in Neighbors)

{

float curDis = Vector3.Distance(item.transform.position , pIn\_Goal.transform.position);

if (curDis < dis && !pIn\_Ignore.Contains(item))

{

dis = curDis;

\_minDisBlock = item;

pIn\_Ignore.Add(item);

}

}

return \_minDisBlock;

}

三中寻路方式总结：

DFS深度优先，纯看脸，找到的路径很瓜皮。

BFS广度优先，缺点，要遍历的对象太多，但找到的一定是最短路径。

Astar，最优解，平均来看是耗时最短的算法，但是找到的路径不一定是最短路径。

如以下情况：

游戏画面

AI 生成的内容可能不正确。

红色为起点，绿色为终点。黑色方格为不可通过区域。

此时黄色为最小代价的位置，但是这条路径却是最远路径。

所以如果需要最短路径则只能选择广度优先搜索。