Федеральное государственное автономное образовательное учреждение высшего образования «Казанский (Приволжский) Федеральный Университет»

Институт вычислительной математики и информационных технологий

Кафедра теоретической кибернетики

КУРСОВАЯ РАБОТА ПО ДИСЦИПЛИНЕ ИНФОРМАТИКА

Тема:

«Алгоритм A-star»

Выполнил: студент 2 курса группы 09-913

дневного отделения ИВМиИТ

Фаизова Алсу Наиловна

Научный руководитель: доцент кафедры

теоретической кибернетики

Байрашева Венера Рустамовна

Казань - 2020

Содержание:

Введение

Достижение пункта назначения по кратчайшему пути – это то, что мы делаем каждый день. A-star (кратко – A*) является одним из наиболее успешных алгоритмов для поиска кратчайшего пути между узлами или графами. Это основанный на имеющейся информации алгоритм поиска, поскольку он использует сведения о стоимости пути и эвристические правила для поиска решения. A* достигает оптимальности и полноты, двух важных свойств для поисковых алгоритмов. Когда алгоритм поиска обладает свойством оптимальности, это означает, что он гарантированно найдет наилучшее из возможных решений. Когда алгоритм поиска обладает свойством полноты, это означает, что если решение данной задачи существует, то алгоритм гарантированно найдет его.

В качестве среды реализации выбрана платформа Unity.

Целями этой работы поставлены следующие:

1. Визуализация алгоритма а стар в 2D;
2. Возможные оптимизации(далее - куча, сплайн, блюр).

Задачи:

1. Разобрать теорию по алгоритму;
2. Написать программу, исполняющий алгоритм и визуально представляющая решение в 2D.

Описание

Алгоритм **A*** — алгоритм поиска, который находит во взвешенном графе маршрут наименьшей стоимости от начальной вершины до выбранной конечной.

В нашем случае взвешенным графом будет карта на плоскости , где за веса отвечает тип препятствия: стена, дорога, трава.

В процессе работы алгоритма для вершин рассчитывается функция f(v)=g(v)+h(v), где

- g(v) — наименьшая стоимость пути в v из стартовой вершины,
- h(v) — эвристическое приближение стоимости пути от v до конечной цели.

Фактически, функция f(v) — длина пути до цели, которая складывается из пройденного расстояния g(v) и оставшегося расстояния h(v). Исходя из этого, чем меньше значение f(v), тем раньше мы откроем вершину v, так как через неё мы предположительно достигнем расстояние до цели быстрее всего. Открытые алгоритмом вершины будем хранить в двоичной куче для оптимизации выбора вершины с наименьшим значение длины фуни f(v).

A* действует подобно алгоритму Дейкстры и просматривает среди всех маршрутов ведущих к цели сначала те, которые благодаря имеющейся информации (эвристическая функция) в данный момент являются наилучшими

В приведенной реализации:

- Q — множество вершин, которые требуется рассмотреть
- U — множество рассмотренных вершин,
- f[x] — значение эвристической функции "расстояние + стоимость" для вершины
- g[x] — стоимость пути от начальной вершины до x,
- h(x) — эвристическая оценка расстояния от вершины x до конечной вершины.

На каждом этапе работы алгоритма из множества Q выбирается вершина `current` с наименьшим значением эвристической функции и просматриваются её соседи. Для каждого из соседей обновляется

расстояние, значение эвристической функции, и он добавляется в множество Q[2].

Псевдокод:

```
bool A*(start, goal):
    U =∅
    Q =∅
    Q.push(start)
    g[start] = 0
    f[start] = g[start] + h(start)
    while Q.size() != 0
        current = вершина из Q с минимальным значением f
        if current == goal
            return true
// нашли путь до нужной вершины
        Q.remove(current)
        U.push(current)
        for v : смежные с current вершины
            tentativeScore = g[current] + d(current, v)
// d(current, v) — стоимость пути между current и v
            if
v∈U
v∈U and tentativeScore >= g[v]
                continue
            if
v∉U
v∉U or tentativeScore < g[v]
                parent[v] = current
                g[v] = tentativeScore
                f[v] = g[v] + h(v)
                if
v∉Q
v∉Q
                    Q.push(v)
    return false
```

Реализация

Реализация основана на пособии [1].

Этап 1. Создание карты в Unity.

Здесь мы в сцену добавляем Plane, Cube, Path(curve imported from Blender), Capsule(Target, Seeker).

- Plane - наша проскость. По умолчанию на ней слой Grass;
- Cube - препятствия;
- Path - кривая, отвечающая за дорогу(задали соответствующее покрытие Road);
- Capsule - объект, которых создаем несколько штук и придаем несколько ролей - Target Seeker - неподвижный пункт назначения и подвижная отправная точка.

Этап 2. Продумывание архитектуры.

Компонентами нашей программы будут

- Class Node - класс элемент сетки - ячейка(вершина взвешенного графа);
- Class Grid - сетка;
- Class Pathfinding - собственно сам алгоритм поиска пути от отправной точки до пункт назначения;
- Class PathRequestManager - обработчик запросов поиска путей;
- Class Unit - ответственный за перемещение отправной объекта(объектов);
- Class Heap - класс бинарная куча.

Class Node

- public bool walkable; - метка проходимости ячейки
- public Vector3 worldPosition; - вектор-позиция ячейки
- public int gridX; - номер ячейки

в сетке по горизонтали

- public int gridY; - номер ячейки

в сетке по вертикали

- public int movementPenalty; - штраф за прохождение ячейк
- public int gCost; - g[x] — стоимость пути от начальной вершины до x,
- public int hCost; - h[x] — эвристическая оценка расстояния от вершины x до конечной вершины.
- public Node parent; - ячейка-предшественник

- int heapIndex; - индекс ячейки в бинарной куче
- public Node(bool _walkable, Vector3 _worldPos, int _gridX, int _gridY, int _penalty); - конструктор
- public int fCost; - f[x] — значение эвристической функции "расстояние + стоимость" для вершины
- public int HeapIndex; - свойство(установить индекс, получить индекс в куче)
- public int CompareTo(Node nodeToCompare); - переопределение метода CompareTo из interface IHeapItem<T> : IComparable<T>

Class Grid

- public bool displayGridGizmos; - отображение размеченной согласно типу покрытия сетки
- public LayerMask unwalkableMask; - слой непроходимого покрытия
- public Vector2 gridWorldSize; - размеры сетки, по которой задана метрика
- public float nodeRadius; - радиус(половина стороны) ячейки сетки
- public TerrainType[] walkableRegions; - массив типов проходимого покрытия
- public int obstacleProximityPenalty = 10; - шраф за близость препятствия
- Dictionary<int, int> walkableRegionsDictionary = new Dictionary<int, int>(); - словарь (тип покрытия - стоимость прохода по нему)
- LayerMask walkableMask; - слой проходимого покрытия
- Node[,] grid; - массив ячеек сетки
- float nodeDiameter; - диаметр(длина стороны) ячейки сетки
- int gridSizeX, gridSizeY; - число ячеек по горизонтали, по вертикали
- int penaltyMin = int.MaxValue; - эти значения понадобятся при градиентной отрисовке
- int penaltyMax = int.MinValue;

- void Awake(); - метод, вызываемый первым
- public int MaxSize; - число ячеек в сетке
- void CreateGrid(); - метод генерации сетки

- void BlurPenaltyMap(int blurSize); - метод применения размытия по Гауссу к сетке
- public List<Node> GetNeighbours(Node node); - метод получения соседей заданной ячейки
- public Node NodeFromWorldPoint(Vector3 worldPosition); - метод получения ячейки по вектору координат
- void OnDrawGizmos(); - метод отрисовки сетки
- public class TerrainType; - класс Тип Поверхности

Class Pathfinding

- PathRequestManager requestManager; - объект типа Обработчик запросов поиска пути(сам класс будет далее)
- Grid grid; - объект Сетка
- void Awake(); метод, вызываемый первым
- public void StartFindPath(Vector3 startPos, Vector3 targetPos); - метод,содержащий StartCoroutine - последовательная обработка кадров сцены и реакция
- IEnumerator FindPath(Vector3 startPos, Vector3 targetPos);
- Vector3[] RetracePath(Node startNode, Node endNode); - массив координат ячеек пути
- Vector3[] SimplifyPath(List<Node> path); - массив координат не всех ячеек, а только где меняется направление
- int GetDistance(Node nodeA, Node nodeB); - расстояние между ячейками(14 единиц - если по диагонали, 10 - по горизонтали и вертикали)

Class Pathfindinrequest

- Queue<PathRequest> pathRequestQueue = new Queue<PathRequest>(); - очередь запросов поиска пути
- PathRequest currentPathRequest; - объект типа PathRequest - запрос поиска пути
- static PathRequestManager instance; - объект для статического метода
- Pathfinding pathfinding; - объект класса Pathfinding
- bool isProcessingPath; - обработывается ли путь

- void Awake(); - метод, запускается первым и генерируется объект класса Pathfinding
- public static void RequestPath(Vector3 pathStart, Vector3 pathEnd, Action<Vector3[], bool> callback); - статический метод, используя два вектора(начальное положение, конечно положение) создается объект типа PathRequest (запрос на поиск и построение пути) и вызывается метод TryProcessNext();
- void TryProcessNext(); - будет запускать поиск пути, если не идёт обработка и пока очередь запросов не пуста
- public void FinishedProcessingPath(Vector3[] path, bool success); -
- struct PathRequest; - структура Запрос поиска пути(поля - начальные координаты, координаты конечные,

Class Unit

- public Transform target; - объект - пункт назначения
- float speed = 20; - скорость передвижения отправного объекта
- Vector3[] path; - путь от отправного объекта до пункта назначения, обновляется(укорачивается) с каждым кадром
- int targetIndex; - индекс обрабатываемой ячейки-элемента пути
- void Start(); - метод запускает обработчик запросов поиска пути
- public void OnPathFound(Vector3[] newPath, bool pathSuccessful); -
- IEnumerator FollowPath(); -
- public void OnDrawGizmos(); - отрисовка

class Heap

- T[] items; - массив типа T
- int currentItemCount; - количество непустых элементов в куче
- public Heap(int maxHeapSize); - конструктор, принимает максимально возможный размер кучи
- public void Add(T item);
- public T RemoveFirst();
- public void UpdateItem(T item);
- public int Count;
- public bool Contains(T item);

- void SortDown(T item);
- void SortUp(T item);
- void Swap(T itemA, T itemB);
- public interface IHeapItem<T> : IComparable<T>;

Листинг

Node.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Node : IHeapItem<Node>
{

    public bool walkable;
    public Vector3 worldPosition;
    public int gridX;
    public int gridY;

    public int movementPenalty;

    public int gCost;
    public int hCost;
    public Node parent;
    int heapIndex;


    public Node(bool _walkable, Vector3 _worldPos, int _gridX, int
_gridY, int _penalty)
    {
        walkable = _walkable;
        worldPosition = _worldPos;
        gridX = _gridX;
        gridY = _gridY;

        movementPenalty = _penalty;
    }

    public int fCost
    {
        get
        {
            return gCost + hCost;
        }
    }

    public int HeapIndex
    {
        get
        {
            return heapIndex;
        }
```

```
        set
        {
            heapIndex = value;
        }
    }

    public int CompareTo(Node nodeToCompare)
    {
        int compare = fCost.CompareTo(nodeToCompare.fCost);
        if (compare == 0)
        {
            compare = hCost.CompareTo(nodeToCompare.hCost);
        }

        return -compare;
    }
}
```

Grid.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Grid : MonoBehaviour
{

    public bool displayGridGizmos;
    public LayerMask unwalkableMask;
    public Vector2 gridWorldSize;
    public float nodeRadius;
    public TerrainType[] walkableRegions;
    public int obstacleProximityPenalty = 10;
    Dictionary<int, int> walkableRegionsDictionary = new
Dictionary<int, int>();
    LayerMask walkableMask; // contains all of the layers in the
walkableRegions array

    Node[,] grid;

    float nodeDiameter;
    int gridSizeX, gridSizeY;

    int penaltyMin = int.MaxValue;
    int penaltyMax = int.MinValue;

    void Awake()
    {
        nodeDiameter = nodeRadius * 2;
        gridSizeX = Mathf.RoundToInt(gridWorldSize.x /
nodeDiameter);
        gridSizeY = Mathf.RoundToInt(gridWorldSize.y /
nodeDiameter);

        foreach (TerrainType region in walkableRegions)
        {
            walkableMask.value |= region.terrainMask.value;

walkableRegionsDictionary.Add((int)Mathf.Log(region.terrainMask.va
lue, 2), region.terrainPenalty);
        }

        CreateGrid();
    }

    public int MaxSize
    {
```

```
        get
        {
            return gridSizeX * gridSizeY;
        }
    }

    void CreateGrid()
    {
        grid = new Node[gridSizeX, gridSizeY];
        Vector3 worldBottomLeft = transform.position -
Vector3.right * gridWorldSize.x / 2 - Vector3.forward *
gridWorldSize.y / 2;

        for (int x = 0; x < gridSizeX; x++)
        {
            for (int y = 0; y < gridSizeY; y++)
            {
                Vector3 worldPoint = worldBottomLeft +
Vector3.right * (x * nodeDiameter + nodeRadius) + Vector3.forward
* (y * nodeDiameter + nodeRadius);
                bool walkable = !(Physics.CheckSphere(worldPoint,
nodeRadius, unwalkableMask));

                int movementPenalty = 0;

                Ray ray = new Ray(worldPoint + Vector3.up * 50,
Vector3.down);
                RaycastHit hit;
                if (Physics.Raycast(ray, out hit, 100,
walkableMask))
                {

walkableRegionsDictionary.TryGetValue(hit.collider.gameObject.laye
r, out movementPenalty);
                }

                if (!walkable)
                {
                    movementPenalty += obstacleProximityPenalty;
                }

                grid[x, y] = new Node(walkable, worldPoint, x, y,
movementPenalty);
            }
        }

        BlurPenaltyMap(3);
```

```
    }

    void BlurPenaltyMap(int blurSize)
    {
        int kernelSize = blurSize * 2 + 1;
        int kernelExtents = blurSize;

        int[,] penaltiesHorizontalPass = new int[gridSizeX,
gridSizeY];
        int[,] penaltiesVerticalPass = new int[gridSizeX,
gridSizeY];

        for (int y = 0; y < gridSizeY; y++)
        {
            for (int x = -kernelExtents; x <= kernelExtents; x++)
            {
                int sampleX = Mathf.Clamp(x, 0, kernelExtents);
                penaltiesHorizontalPass[0, y] += grid[sampleX,
y].movementPenalty;
            }

            for (int x = 1; x < gridSizeX; x++)
            {
                int removeIndex = Mathf.Clamp(x - kernelExtents -
1, 0, gridSizeX);
                int addIndex = Mathf.Clamp(x + kernelExtents, 0,
gridSizeX - 1);

                penaltiesHorizontalPass[x, y] =
penaltiesHorizontalPass[x - 1, y] - grid[removeIndex,
y].movementPenalty + grid[addIndex, y].movementPenalty;
            }
        }

        for (int x = 0; x < gridSizeX; x++)
        {
            for (int y = -kernelExtents; y <= kernelExtents; y++)
            {
                int sampleY = Mathf.Clamp(y, 0, kernelExtents);
                penaltiesVerticalPass[x, 0] +=
penaltiesHorizontalPass[x, sampleY];
            }

            int blurredPenalty =
Mathf.RoundToInt((float)penaltiesVerticalPass[x, 0] / (kernelSize
* kernelSize));
            grid[x, 0].movementPenalty = blurredPenalty;
```

```
            for (int y = 1; y < gridSizeY; y++)
            {
                int removeIndex = Mathf.Clamp(y - kernelExtents -
1, 0, gridSizeY);
                int addIndex = Mathf.Clamp(y + kernelExtents, 0,
gridSizeY - 1);

                penaltiesVerticalPass[x, y] =
penaltiesVerticalPass[x, y - 1] - penaltiesHorizontalPass[x,
removeIndex] + penaltiesHorizontalPass[x, addIndex];
                blurredPenalty =
Mathf.RoundToInt((float)penaltiesVerticalPass[x, y] / (kernelSize
* kernelSize));
                grid[x, y].movementPenalty = blurredPenalty;

                if (blurredPenalty > penaltyMax)
                {
                    penaltyMax = blurredPenalty;
                }
                if (blurredPenalty < penaltyMin)
                {
                    penaltyMin = blurredPenalty;
                }
            }
        }

    }

    public List<Node> GetNeighbours(Node node)
    {
        List<Node> neighbours = new List<Node>();

        for (int x = -1; x <= 1; x++)
        {
            for (int y = -1; y <= 1; y++)
            {
                if (x == 0 && y == 0)
                    continue;

                int checkX = node.gridX + x;
                int checkY = node.gridY + y;

                if (checkX >= 0 && checkX < gridSizeX && checkY >=
0 && checkY < gridSizeY)
                {
                    neighbours.Add(grid[checkX, checkY]);
                }
            }
```

```
        }

        return neighbours;
    }


    public Node NodeFromWorldPoint(Vector3 worldPosition)
    {
        float percentX = (worldPosition.x + gridWorldSize.x / 2) /
gridWorldSize.x;
        float percentY = (worldPosition.z + gridWorldSize.y / 2) /
gridWorldSize.y;
        percentX = Mathf.Clamp01(percentX);
        percentY = Mathf.Clamp01(percentY);

        int x = Mathf.RoundToInt((gridSizeX - 1) * percentX);
        int y = Mathf.RoundToInt((gridSizeY - 1) * percentY);
        return grid[x, y];
    }

    void OnDrawGizmos()
    {
        Gizmos.DrawWireCube(transform.position, new
Vector3(gridWorldSize.x, 1, gridWorldSize.y));
        if (grid != null && displayGridGizmos)
        {
            foreach (Node n in grid)
            {

                Gizmos.color = Color.Lerp(Color.white,
Color.black, Mathf.InverseLerp(penaltyMin, penaltyMax,
n.movementPenalty));
                Gizmos.color = (n.walkable) ? Gizmos.color :
Color.red;
                Gizmos.DrawCube(n.worldPosition, Vector3.one *
(nodeDiameter));
            }
        }
    }

    [System.Serializable]
    public class TerrainType
    {
        public LayerMask terrainMask;
        public int terrainPenalty;
    }
```

}

Pathfinding.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Diagnostics;
using System;

public class Pathfinding : MonoBehaviour
{

    PathRequestManager requestManager;
    Grid grid;

    void Awake()
    {
        requestManager = GetComponent<PathRequestManager>();
        grid = GetComponent<Grid>();
    }


    public void StartFindPath(Vector3 startPos, Vector3 targetPos)
    {
        StartCoroutine(FindPath(startPos, targetPos));
    }

    IEnumerator FindPath(Vector3 startPos, Vector3 targetPos)
    {
        Stopwatch sw = new Stopwatch();
        sw.Start();

        Vector3[] waypoints = new Vector3[0];
        bool pathSuccess = false;

        Node startNode = grid.NodeFromWorldPoint(startPos);
        Node targetNode = grid.NodeFromWorldPoint(targetPos);

        if (startNode.walkable && targetNode.walkable)
        {
            Heap<Node> openSet = new Heap<Node>(grid.MaxSize);
            HashSet<Node> closedSet = new HashSet<Node>();
            openSet.Add(startNode);

            while (openSet.Count > 0)
            {
                Node node = openSet.RemoveFirst();
                closedSet.Add(node);

                if (node == targetNode)
```

```
                {
                    sw.Stop();
                    print("Path found: " + sw.ElapsedMilliseconds
+ "ms");
                    pathSuccess = true;
                    break;
                }

                foreach (Node neighbour in
grid.GetNeighbours(node))
                {
                    if (!neighbour.walkable ||
closedSet.Contains(neighbour))
                    {
                        continue;
                    }

                    int newCostToNeighbour = node.gCost +
GetDistance(node, neighbour) + neighbour.movementPenalty;
                    if (newCostToNeighbour < neighbour.gCost ||
!openSet.Contains(neighbour))
                    {
                        neighbour.gCost = newCostToNeighbour;
                        neighbour.hCost = GetDistance(neighbour,
targetNode);
                        neighbour.parent = node;

                        if (!openSet.Contains(neighbour))
                            openSet.Add(neighbour);
                        else
                        {
                            // if it is already in the openset
that it means this value changed
                            openSet.UpdateItem(neighbour);
                        }
                    }
                }
            }
        }


        yield return null;                  //wait for one frame
before returning
        if (pathSuccess)
        {
            waypoints = RetracePath(startNode, targetNode);
        }
```

```
        requestManager.FinishedProcessingPath(waypoints,
pathSuccess);
    }

    Vector3[] RetracePath(Node startNode, Node endNode)
    {
        List<Node> path = new List<Node>();
        Node currentNode = endNode;

        while (currentNode != startNode)
        {
            path.Add(currentNode);
            currentNode = currentNode.parent;
        }

        Vector3[] waypoints = SimplifyPath(path);
        Array.Reverse(waypoints);
        return waypoints;
    }

    Vector3[] SimplifyPath(List<Node> path)
    {
        // Simplify the path
        // So the waypoints are only placed wherever the path
actually changes direction
        // so we don't have hundreds of sort of superfluous
waypoints

        List<Vector3> waypoints = new List<Vector3>();
        Vector2 directionOld = Vector2.zero; // stores the
direction of last two nodes

        for (int i = 1; i < path.Count; i++)
        {
            Vector2 directionNew = new Vector2(path[i - 1].gridX -
path[i].gridX, path[i - 1].gridY - path[i].gridY);
            if (directionNew != directionOld) // if path changed
direction (between neighbours in the path - up/up-tha same
direction, up/right - the different)
            {

waypoints.Add(path[i].worldPosition);//waypoints.Add(path[i-1].wor
ldPosition);
            }
            directionOld = directionNew;
        }

        return waypoints.ToArray();
```

```
    }

    int GetDistance(Node nodeA, Node nodeB)
    {
        int dstX = Mathf.Abs(nodeA.gridX - nodeB.gridX);
        int dstY = Mathf.Abs(nodeA.gridY - nodeB.gridY);

        if (dstX > dstY)
            return 14 * dstY + 10 * (dstX - dstY);
        return 14 * dstX + 10 * (dstY - dstX);
    }
}
```

PathRequestManager.cs

```
using System.Collections;
using System.Collections.Generic;// to import Queue
using UnityEngine;
using System; // to import Action


public class PathRequestManager : MonoBehaviour
{

    Queue<PathRequest> pathRequestQueue = new
Queue<PathRequest>();
    PathRequest currentPathRequest;

    static PathRequestManager instance;
    Pathfinding pathfinding;

    bool isProcessingPath;

    void Awake()
    {
        instance = this;
        pathfinding = GetComponent<Pathfinding>();
    }

    public static void RequestPath(Vector3 pathStart, Vector3
pathEnd, Action<Vector3[], bool> callback)
    {
        PathRequest newRequest = new PathRequest(pathStart,
pathEnd, callback);
        instance.pathRequestQueue.Enqueue(newRequest);
        instance.TryProcessNext();
    }

    void TryProcessNext()
    {
        if (!isProcessingPath && pathRequestQueue.Count > 0)
        {
            currentPathRequest = pathRequestQueue.Dequeue();
            isProcessingPath = true;

pathfinding.StartFindPath(currentPathRequest.pathStart,
currentPathRequest.pathEnd);
        }
    }

    public void FinishedProcessingPath(Vector3[] path, bool
success)
```

```
    {
        currentPathRequest.callback(path, success);
        isProcessingPath = false;
        TryProcessNext();
    }

    struct PathRequest
    {
        public Vector3 pathStart;
        public Vector3 pathEnd;
        public Action<Vector3[], bool> callback;

        public PathRequest(Vector3 _start, Vector3 _end,
Action<Vector3[], bool> _callback)
        {
            pathStart = _start;
            pathEnd = _end;
            callback = _callback;
        }

    }
}
```

Heap.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;

public class Heap<T> where T: IHeapItem<T>
{
    //массив типа T
    T[] items;
    // количество непустых элементов в куче
    int currentItemCount;
    // конструктор, принимает максимально возможный размер кучи
    public Heap(int maxHeapSize)
    {
//  создали массив items типа T размера maxHeapSize
        items = new T[maxHeapSize];
    }
// метод для добавления элемента в кучу
    public void Add(T item)
    {
        // индекс добавляемого элемента приравнивается числу,
равному количеству непустых элементов в куче(согласно реализации
кучи)
        item.HeapIndex = currentItemCount;
       // добавляем этот элемент в кучу(в массив, хранящий
элементы кучи)
        items[currentItemCount] = item;
     // проверим, не нарушилась ли куча, и уладим, чтоб её
свойства сохранялись, вызвав метод SortUp(item)
        SortUp(item);
     // увеличиваем переменную-счетчик количества непустых
элементов в куче
        currentItemCount++;
    }
// метод удаляет первый элемент в массиве кучи, возвращая его
    public T RemoveFirst()
    {
     // получаем первый элемент из массива кучи
        T firstItem = items[0];
     // уменьшаем переменную, равную количеству непустых элементов
в куче
        currentItemCount--;
     // в качестве первого элемента(корня) кучи вставляем элемент
с наибольшим индексом(он равен currentItemCount)
        items[0] = items[currentItemCount];
     // меняем индекс этого элемента на 0(значит, что он в корне)
        items[0].HeapIndex = 0;
```

```
     // после этой операции свойство кучи могло нарушиться -
вызываем метод SortDown(items[0]), параметром передаем перешедший
в корень элемент
        SortDown(items[0]);
     // возвращаем бывший первый элемент(корень) кучи
        return firstItem;

    }

    public void UpdateItem(T item) // if path with lower cost
found
    {
        SortUp(item);
    }

    public int Count
    {
        get
        {
            return currentItemCount;
        }
    }

    public bool Contains(T item)
    {
        return Equals(items[item.HeapIndex], item);
    }
// метод восстанавливает свойства кучи после извлечения из неё
первого элемента, где item теперь первый(корневой)
    void SortDown(T item)
    {
        while(true)
        {
// рассчитываем согласно формуле индексы дочерних к item элементов
            int childIndexLeft = item.HeapIndex * 2 + 1;
            int childIndexRight = item.HeapIndex * 2 + 2;
            int swapIndex = 0;

// выбираем левый дочерний элемент, запоминаем его индекс в
swapIndex. Если оказалось, что у значения правого дочернего
элемента выше приоритет(меньше значение целевой функции), чем у
левого, то swapIndex будет хранить его индекс. В дальнейшем оценим
приоритет текущего элемента item и элемента с индексом swapIndex
            if (childIndexLeft < currentItemCount)
            {
                swapIndex = childIndexLeft;

                if (childIndexRight < currentItemCount)
```

```
                {
                    if
(items[childIndexLeft].CompareTo(items[childIndexRight]) < 0)
                    {
                        swapIndex = childIndexRight;
                    }
                }
     // если приоритет swapIndex выше(значение целевой функции
ниже), то меняем его с текущим элементом item
                if (item.CompareTo(items[swapIndex]) < 0)
                {
                    Swap(item, items[swapIndex]);

                }
                else
                {
                    return;
                }
            }
// если у текущей вершины нет дочерних элементов
            else             {
                return;
            }
        }
    }

    void SortUp(T item)
    {
      // получаем индекс родительской к текущему элементу
item(согласно формуле)
        int parentIndex = (item.HeapIndex - 1) / 2;

        while(true)
        {
            T parentItem = items[parentIndex];
     // сравниваем текущий элемент с родительским, используя метод
CompareTo() класса Node (CompareTo() возвращает 1, если у item
выше приоритет(Согласно реализации - ниже значение параметра
сравнения, то есть целевой функции), чем у parentItem, 0 - если у
них равные приоритеты, и -1 - у parentItem выше(ниже значение
параметра сравнения), чем у item)
// итак, если у item выше приоритет в куче(ниже значение параметра
сравнения) - обмениваем с parentItem
            if (item.CompareTo(parentItem) > 0)
            {
     // метод меняет элементы в массиве и обменивает индексы
элементов в куче
                Swap(item, parentItem);
```

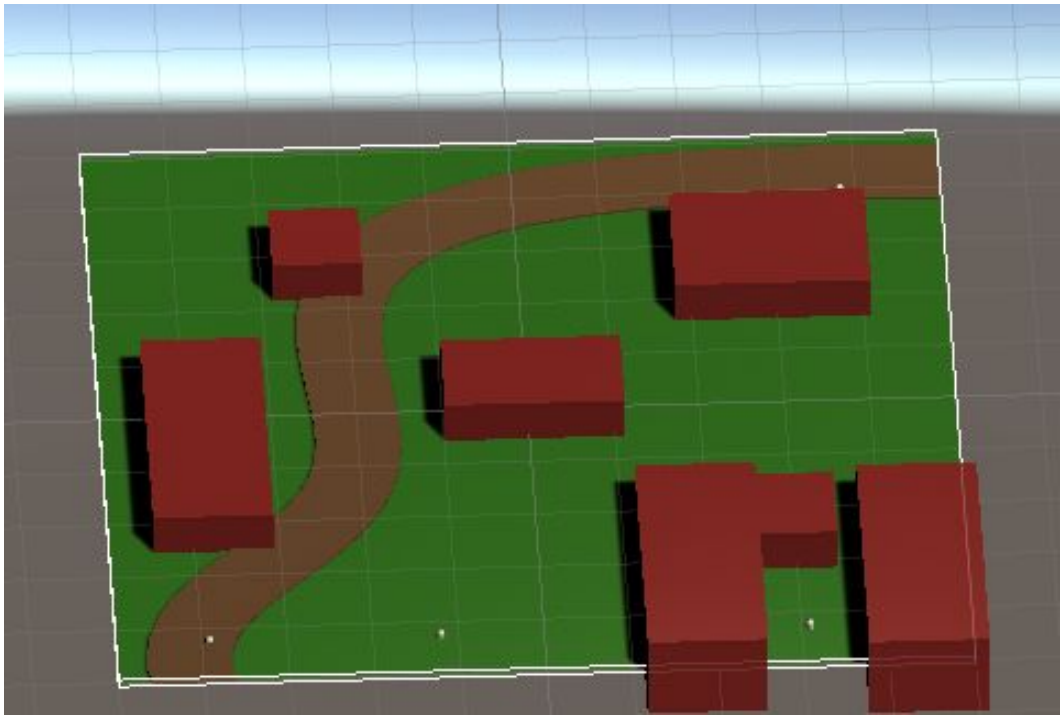
```
            }
            else
            {
                break;
            }
// пересчитывает индекс родительский индекс(если произошел обмен, текущий элемент изменил свою позицию, но свойство кучи может всё ещё быть нарушено)
            parentIndex = (item.HeapIndex - 1) / 2;
        }
    }
     // метод меняет элементы в массиве и обменивает индексы элементов в куче
    void Swap(T itemA, T itemB)
    {
        items[itemA.HeapIndex] = itemB;
        items[itemB.HeapIndex] = itemA;
        int itemAIndex = itemA.HeapIndex;
        itemA.HeapIndex = itemB.HeapIndex;
        itemB.HeapIndex = itemAIndex;
    }
}

public interface IHeapItem<T> : IComparable<T>
{
    int HeapIndex
    {
        get;
        set;
    }
}
```

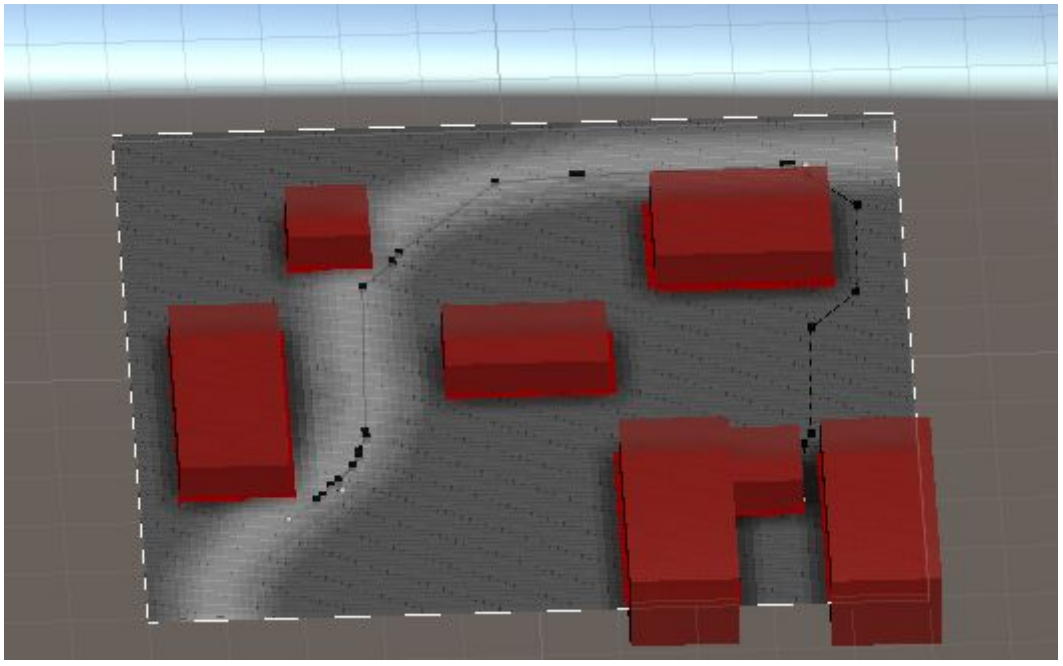
Unit.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Unit : MonoBehaviour
{
    public Transform target;
    float speed = 20;
    Vector3[] path;
    int targetIndex;

    void Start()
    {
        PathRequestManager.RequestPath(transform.position,
target.position, OnPathFound);
    }

    public void OnPathFound(Vector3[] newPath, bool
pathSuccessful)
    {
        if (pathSuccessful)
        {
            path = newPath;
            targetIndex = 0;
            StopCoroutine("FollowPath");
            StartCoroutine("FollowPath");
        }
    }

    IEnumerator FollowPath()
    {
        Vector3 currentWaypoint = path[0];
        while (true)
        {
            if (transform.position == currentWaypoint) // so it is
need to advance to the next waypoint
            {
                targetIndex++; // targerIndex of the waypoint it
the path
                if (targetIndex >= path.Length)
                {
                    yield break; // exit out of a coroutine
                }
                currentWaypoint = path[targetIndex];
            }
            // Each frame there will be move(if the path is found)
out transform closer to the current waypoint
```

```
            // Passing in a current position and a target
position(currentWaypoint)
            transform.position =
Vector3.MoveTowards(transform.position, currentWaypoint, speed *
Time.deltaTime);
            yield return null; // to move over to the next frame

        }
    }

    public void OnDrawGizmos()
    {
        if (path != null)
        {
            for (int i = targetIndex; i < path.Length; i++)
            {
                Gizmos.color = Color.black;
                Gizmos.DrawCube(path[i], Vector3.one);

                //Draw lines to connect up waypoints
                if (i == targetIndex)
                {
                    // draw line from current position to path[i]
                    Gizmos.DrawLine(transform.position, path[i]);
                }
                else
                {
                    Gizmos.DrawLine(path[i - 1], path[i]);
                }
            }
        }
    }
}
```

Демонстрация работы





Итоги

В ходе работы была разобрана теория, изучены возможные пути реализации алгоритма, а также некоторые методы оптимизации(класс Heap - куча для эффективной обработки данных). Далее спроектирована архитектура и осуществлена программная реализация.
Вдобавок к поставленным задачам, исследован вопрос о поиске пути в 3D[3-7].

Список использованных ресурсов

1. A* Pathfinding Tutorial[Электронный ресурс] : YouTube - Режим доступа: [https://www.youtube.com/playlist?list=PLFt_AvWsXl0cq5Umv3pMC9SPnKjfp9eGW] (Дата обращения 23.11.2020)
2. Алгоритм A* [Электронный ресурс] : Виконспекты ИТМО - Режим доступа: [https://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_A*] (Дата обращения 13.11.2020)
3. Mercuna 3D Navigation[Электронный ресурс] - Режим доступа: [https://mercuna.com/3d-navigation/] (Дата обращения 25.11.2020)
4. Mercuna 3D Navigation Smooth Spline Path Following [Электронный ресурс] : YouTube - Режим доступа: [https://www.youtube.com/watch?v=lL-NJMRMyDo] (Дата обращения 25.11.2020)
5. UE4 AI Pathfinding Solution[Электронный ресурс] : YouTube - Режим доступа: [https://www.youtube.com/watch?v=nyRbqI5u9aM] (Дата обращения 25.11.2020)
6. My experience with Pathfinding (JPS + A*) in Unity[Электронный ресурс] : YouTube - Режим доступа: [https://www.youtube.com/watch?v=yAwPfvK6_6E&t=511s] (Дата обращения 25.11.2020)
7. 3D A* Pathfinding for Unity[Электронный ресурс] : GitHub - Режим доступа: [https://github.com/CBlan/3D_A-Star_Pathfinding] (Дата обращения 26.11.2020)