A lightweight tile based A-Star implementation

1 Version

16.10.2016 v 1.0.2 initial release

2 Introduction

Welcome and thank you for purchasing our A star library.

The A star algorithm is a pathfinding and graph traversal algorithm widely used in video games. Basically it can find the ‘cheapest’ path through a graph, knowing the ‘costs’ required to travel between nodes.

This is a quick guide will show you how to get started.

There are **2 core components** used for pathfinding:

T**he first component** is **Grid3di**. It is used to hold your map.
The second component is **AStarAgent** . It is the agent that can travel inside your map.

A third component we created for demonstration purposes is a simple tile map class, to render the content of our map to the screen. It is only used for visualization in the demo scenes! It can be replaced by any full blown tile engine later. Everything else is build around those 2 components and not really required.

Together with the sample scenes in the project everything is very easy to understand, and the source code is well commented.

3 The Grid3di class

Grid3di is a very simple helper class that allows you to hold a 2 or 3 dimensional array of integers. This map holds the structure of your maze or map. The value of every cell represents the costs to cross that cell.

To construct a new tilemap we write:

```
Grid3di grid = new Grid3di(32,16)
```

A new tilemap with the dimensions of 32x16 cells is constructed with only layer (depth =1). Every cell will be filled with 0.

There are convenience functions that allow you to parse a `Grid3di` from strings:

```
string s_maze_layer_0 =
                  "################################"+
                  "# #  #          #       #      #"+
                  "# #  #          #       #      #"+
                  "# #  #  ###aaaa #   # # #  ### #"+
                  "# #  #    #a    #   # #a#  # # #"+
                  "# #  #    #a aaa#   #   #  # # a"+
                  "# #  #    #a a a#   #a# #  # # #"+
                  "# #  #    #a aaa#   # # #  # # #"+
                  "# #  #    #a    #   #   #  a # #"+
                  "# #  #    #a    #   #   #  a # #"+
                  "# #  ###  #######   # #a#  # # a"+
                  "# #    #      aa#   # # #  # a #"+
                  "# #    #  ###  ###  #      # # #"+
                  "# #         aa  #   ######## a #"+
                  "# #         aa  #            a #"+
                  "################################";

Dictionary<char,int> ValueLookup = new Dictionary<char,int>();

ValueLookup['#'] = 99;
ValueLookup[' '] = 1;
ValueLookup['a'] = 10;

srcGrid.ParseLayerFromString(s_maze_layer_0,ValueLookup,0);
srcGrid.BorderCode = 99;
```

The Value lookup dictionary tells the function `ParseLayerFromString` which code to write for which character in the string. In our sample the tilemap used for visualization can create the **`ValueLookup`** dictionary on the fly.

In our example '#' represents a wall that can not be passed. ' ' represents empty space with the cost of 1 to travel along (default). 'A' represents swamp or a pit with the cost of 10 to travel along. You will see in the demonstration how the algorithm tries to avoid those fields, until the costs of walking around become higher than the cost to cross them.

Now we have a map. Half of the work is done.

4 The AStarAgent class.

In pathfinding you need a maze and an agent that travels within the maze.The AStarAgent represents that agent. It can travel autonomously inside a map once given a start and end coordinate.

It is derived from MonoBehaviour, because it uses a coroutine for travelling inside the map.

Inside your class you can write:

```
public AStarAgent agent;

Vec3di pathStart = new Vec3di( 1,14,0);
Vec3di pathEnd   = new Vec3di(30, 1,0);

agent.map = srcGrid;
agent.ownPosition = pathStart;
agent.FindPath( pathEnd, curPath,AllowDiagonals);
```

Now the agent travels along the found path with the given speed parameters. Everything you have to do is to update the local coordinates and euler angles of the game object representing your agent.

5 A sample implementation

In the example scenes we have used a simple tilemap class to render our maps. Tilemaps were used in classic video games that had small amount of video memory. Typically one byte was representing one tile. For demonstration purpose our tilemap only renders colored rectangles. The sample tilemaps can be configured in the Unity Editor. Everything you have to do is to configure the sprites you want to be rendered for the map codes. You can specify a custom color for every sprite. In our example we use the same sprite for all the map tiles, we only use different colors. There is lots of space for improvements here ;-)

So here is a complete listing for a demo class. It can be attached to a GameObject and configured in the unity editor.

```
public class AStarDemo : MonoBehaviour
{
    protected Vec3di pathStart=new Vec3di();
    protected Vec3di pathEnd = new Vec3di();

    /// <summary>
    /// holds the current path between path Start and Path End
```

```
/// </summary>
public List<Vec3di> curPath = new List<Vec3di>();

// the width height and depth of the map
public int mapWidth=32;
public int mapHeight=16;
public int mapDepth=1;

// our map
protected Grid3di srcGrid;

// another map we use to visualize information aboutthe current path
Grid3di resGrid;

// tilemap used to render the maze
public  Tilemap tilemaps;

// tilemap used to render the current path
Public Tilemap tilemaps_path;

/// <summary>
/// a maze agent that is capable of moving in a maze
/// </summary>
public AStarAgent agent;

// a gameobject representing the agent
public GameObject agentRep;

// toggle diagonal maze movement on/off
public bool AllowDiagonals = false;

bool pathNeedsUpdate = true;

Vector3 screenPos;
Vec3di lastTilePos = new Vec3di();

void InitializeMaze()
{
  string s_maze_layer_0 =
              "################################"+
              "# #  #          #       #      #"+
              "# #  #          #       #      #"+
              "# #  #  ###aaaa #   # # #  ### #"+
              "# #  #    #a    #   # #a#  # # #"+
              "# #  #    #a aaa#   #   #  # # a"+
              "# #  #    #a a a#   #a# #  # # #"+
              "# #  #    #a aaa#   # # #  # # #"+
              "# #  #    #a    #   #   #  a # #"+
              "# #  #    #a    #   #   #  a # #"+
              "# #  ###  #######   # #a#  # # a"+
              "# #    #      aa#   # # #  # a #"+
              "# #    #  ###  ###  #      # # #"+
              "# #         aa  #   ######## a #"+
              "# #         aa  #            a #"+
              "################################";

    srcGrid.ParseLayerFromString(s_maze_layer_0,tilemaps.ValueLookup,0);
    srcGrid.BorderCode=99;

    pathStart = new Vec3di(1,1,0);
    pathEnd = new Vec3di(1,1,0);
```

```
     agent.ownPosition = pathStart;
}

void Start()
{
    srcGrid = new Grid3di();
    srcGrid.Resize(mapWidth, mapHeight, mapDepth);

    resGrid = new Grid3di(mapWidth,mapHeight,mapDepth);

    agent.map = srcGrid;
    agent.ownPosition = pathStart;

    agentRep.gameObject.SetActive(true);

    InitializeMaze();

   pathNeedsUpdate = true;

}

void Update()
{
    // get local position within AStarTest object
    screenPos = Camera.main.ScreenToWorldPoint(Input.mousePosition);

    // calculate tile position
    Vec3di tilePosMouse = tilemaps_path.LocalPosToTilePos(
                  tilemaps_path.transform.InverseTransformPoint(screenPos));

    if(pathNeedsUpdate)
    {
           // try to find a new path
           agent.FindPath( pathEnd, curPath,AllowDiagonals);

           tilemaps.ApplyMap(srcGrid,0);

           agent.Follow(curPath);

          pathNeedsUpdate=false;
    }

    resGrid.Clear();

   // we have configured the codes 9,10, and 11 in the tilemaps_path in the editor
   // to render the current path!
    ApplySolutionToMap(resGrid,9);

    resGrid.SetAt(pathStart.x, pathStart.y, pathStart.z, 10);
    resGrid.SetAt(pathEnd.x,   pathEnd.y,   pathEnd.z,   10);
    resGrid.SetAt(tilePosMouse.x,  tilePosMouse.y, 0, 11);

    tilemaps_path.ApplyMap(resGrid, 0);

    agentRep.transform.localPosition = tilemaps_path.TilePosToLocalPos(
           agent.ownPosition.x, agent.ownPosition.y, agentRep.transform.localPosition.z);

    agentRep.transform.eulerAngles = new Vector3(0, 0, agent.curAimXYAnimated);

   // test if the mouse has moved to a different tile
    if (lastTilePos.Compare(tilePosMouse) == false)
    {
```

```
            if (srcGrid.GetAt(tilePosMouse) != srcGrid.BorderCode)
            {
                agent.Stop();
                pathEnd = tilePosMouse;
                pathStart = agent.ownPosition;
                pathNeedsUpdate=true;

                lastTilePos = tilePosMouse;
            }
        }
    }


    public void ApplySolutionToMap(Grid3di map , int code )
    {
        foreach( Vec3di step in curPath )
        {
            map.SetAt(step.x,step.y,step.z,code);
        }
    }
}
```

6 Helper classes

We have included some other useful classes for the Demonstration scenes. Those classes are only for visualisation purposes and are not required by the 2 core components (Grid3di and `AStarAgent`)

6.1 The maze class

The maze class allows to create random mazes like shown on *https://en.wikipedia.org/wiki/Maze_generation_algorithm*
The maze class can further be used to initialize a Grid3di instance with it's content.

6.2 The tilemaps class

We have included a very simple tilemap class in the sample scenes.
The purpose was only to visualize the map we are using for pathfinding and to show how the agent travels in those maps. You can use your own tilemap implementation, this class here only demonstrates that the dimensions of the tilemaps for visualisation have to correlate with the Grid3di instance.

6.3 The button class

A very simple button class, only used in the demo scenes.