Pathfinding

A* algorithm

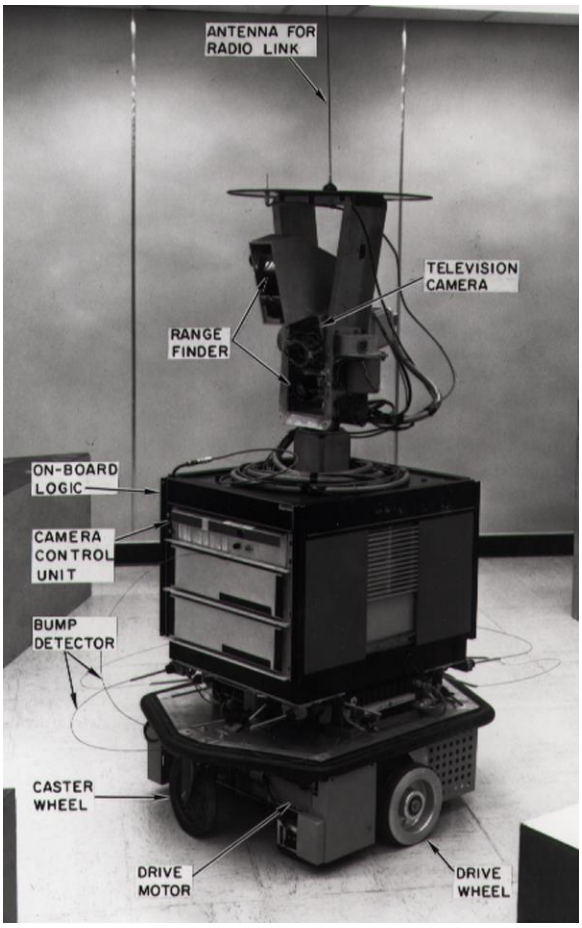
Unity3D Project

Content

In this project, I will create and explain the A* pathfinding algorithm using Unity 3D Engine. Pathfinding is related to the shortest path problem which examines how to identify the path that best meets some criteria (shortest, cheapest) between two points.

There are multiple algorithms that solve this problem each with its own pros and cons. In this project, I will be using the A* pathfinding algorithm due to its optimality and completeness.

A* was created as part of the *Shakey project* in 1968*.,* which had the aim of building a mobile robot that could own actions. A* was originally used for finding optimal least-cost paths when the cost of a path is the sum of its costs, but it has been shown that A* can be used to find optimal paths for any problem satisfying the conditions of a cost-algebra.



Picture 1) Shakey the robot – year of creation: 1966-1972

Shakey the robot was the first general-purpose mobile robot to be able to reason about its own actions. While other robots would have to be instructed on each individual step of completing a larger task, Shakey could analyze commands and break them into basic chunks by itself.

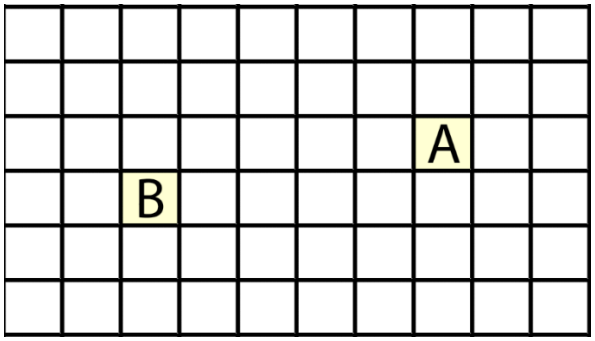
A-star algorithm

A* algorithm is an informed search algorithm or a best-first search. It is one of the best and popular algorithms used in path-finding and graph traversals. A* algorithm is really a smart algorithm that separates it from other conventional algorithms.

What A* search algorithm does is that at each step it picks the node according to a value ***f*** which is a parameter equal to the sum of two other parameters ***g*** and ***h.*** At each step it picks the node/cell having the lowest ***f*** and process that node/cell.

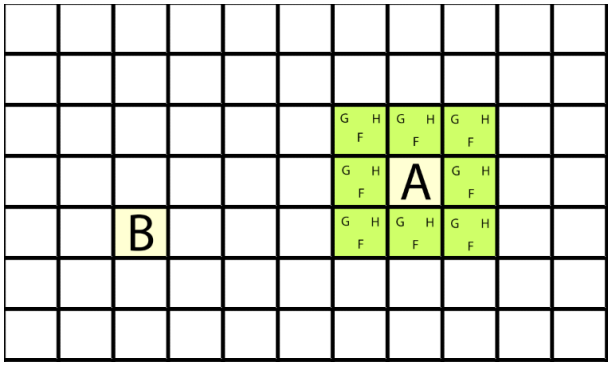
g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there

h = the estimated movement cost to move from that given square on the grid to the final destination.



Picture 2) A to B point grid

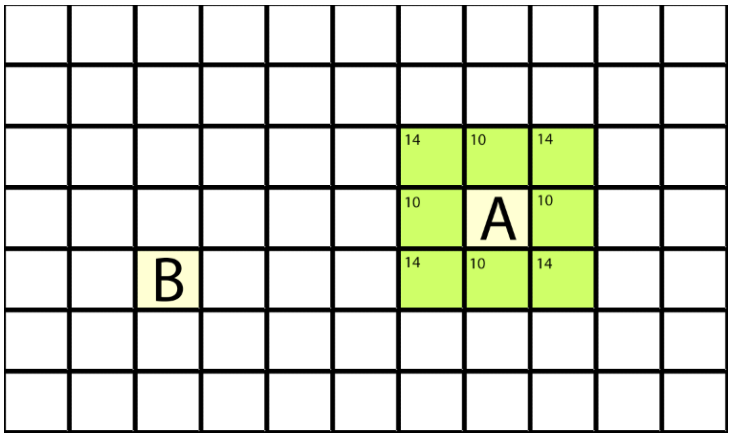
Let's say we have point A and point B. Adding ***f, g*** and ***h*** arguments we can see how they are placed on the grid below.



Picture 3) Arguments added

The algorithm begins by going to the starting node, node **A** and looking at all of its surrounding nodes and calculating some values for each of them.

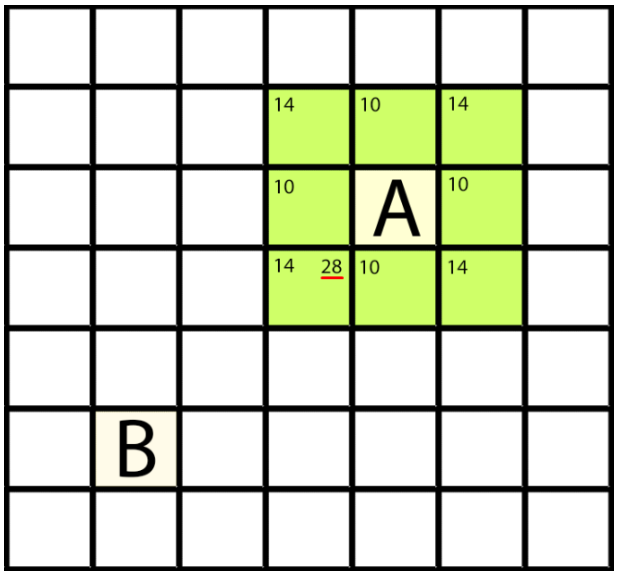
G cost is the value in the top left corner of each node. That is how far away that node is from the starting node.



Picture 4) Value 14 added to the G cost

The node in the top left corner has got a **G** cost of 14 since it is just one diagonal move away from node **A,** value 10 is added to the top, left, right and bottom node.

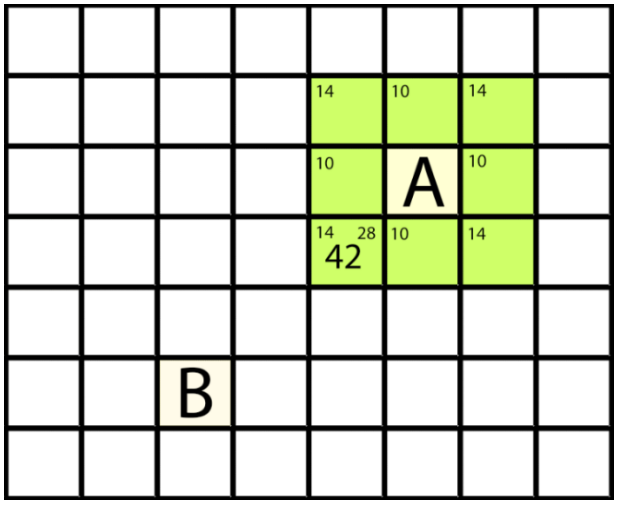
For better understanding, I moved point B more to the bottom left, but nothing else changed. Anyways, in the top right corner of each node is the **H** cost, which is the opposite of **G** cost. It is how far away is from the end node (point **B**).



Picture 5) H cost - distance from the end node

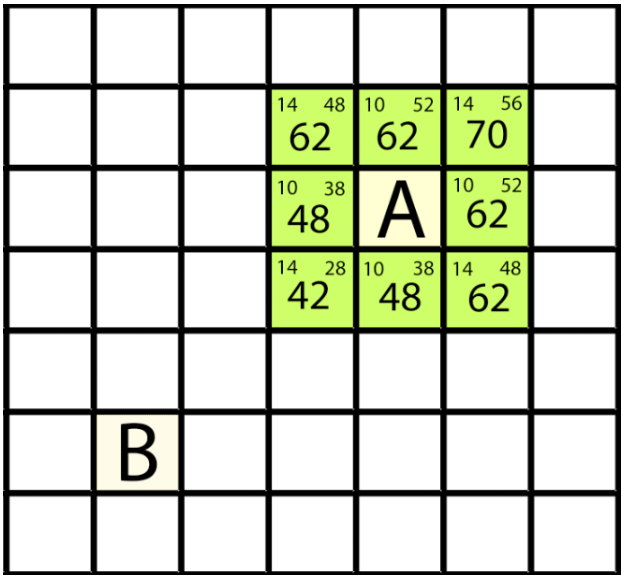
In this case, this node is two diagonal moves away from the end node, so it is got an **H** cost of 28.

Now, the **F** cost is just a sum of **G** and **H** cost. In this case, the value of **F** cost would be 42.



Picture 6) F cost value = G cost + H cost

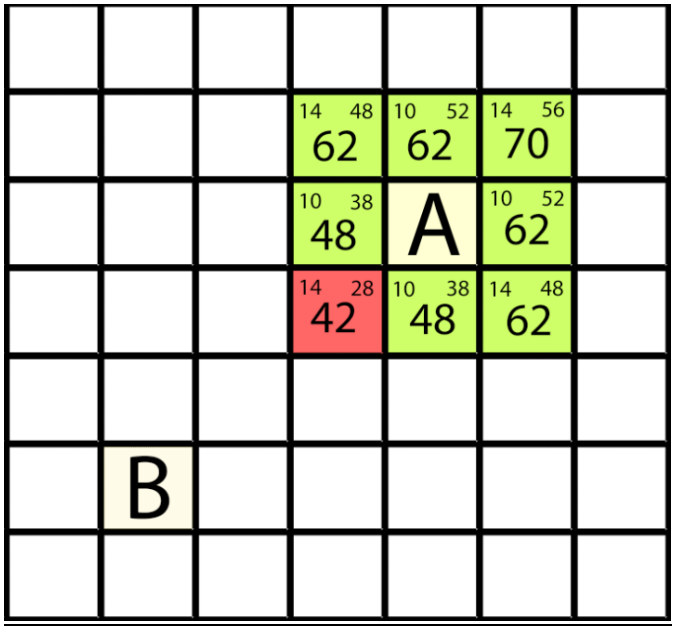
Following this method, with each value-added, the algorithm is going to look at the node with the lowest **F** cost.



Picture 7) G, H and F values added to the grid

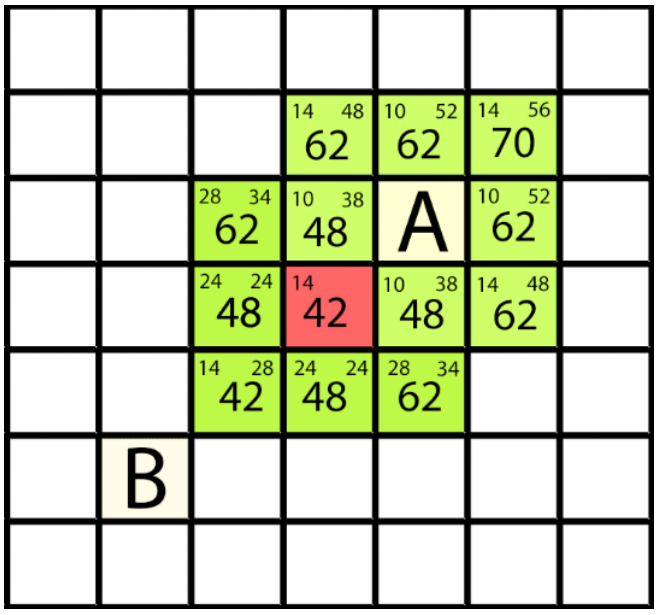
That is the value in the bottom left corner 42.

Now the algorithm is marking this as *closed* and the square appears red.



Picture 8) Value 42 marked as closed

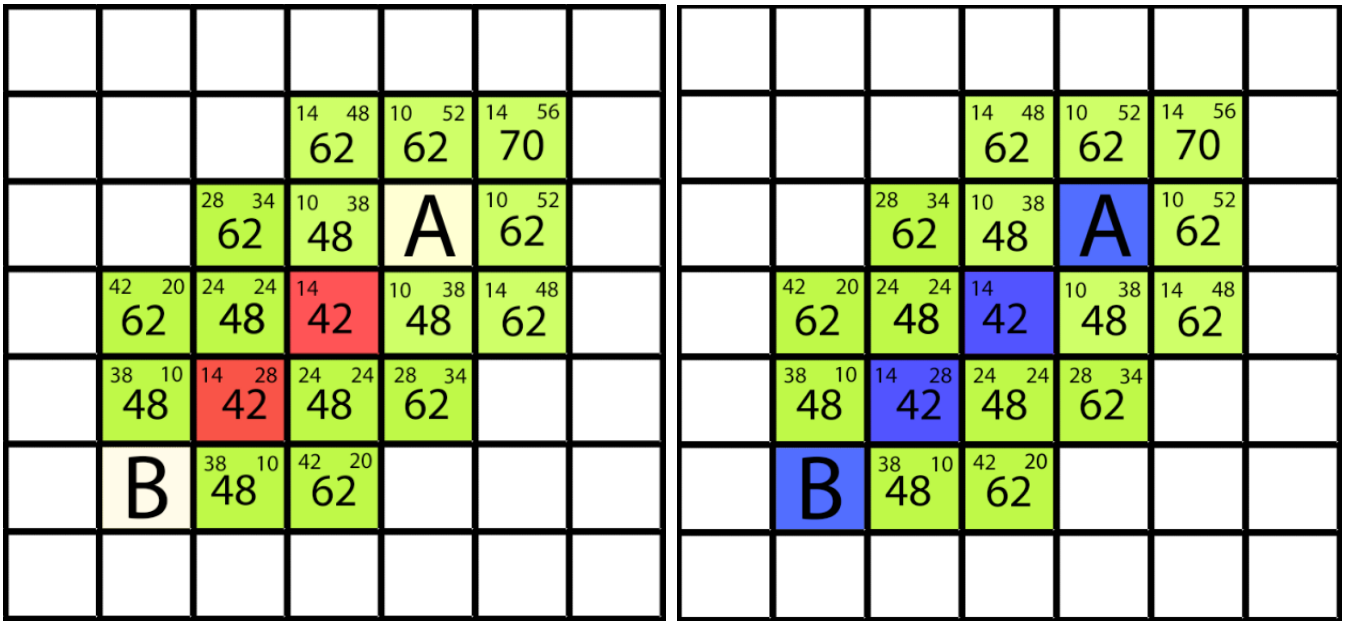
Following this logic, the next node will calculate all of its surrounding nodes again until it finds the endpoint **B.**



Picture 9) New node

It is quite simple to see how the logic behind the A* algorithm works. The **F** costs are staying the same as the path moves towards the endpoint **B** because the **G** costs (movement costs) are increasing, and the **H** costs are decreasing.

Continuing this method we get the final path that looks something like this:



Picture 10) Clear path between point A and point B

Now that we have a brief understanding of the A* pathfinding algorithm, let us take a look at the pseudocode and how the whole package works.

```
OPEN //the set of nodes to be evaluated
CLOSED //the set of nodes already evaluated
add the start node to OPEN

loop
 current = node in OPEN with the lowest f_cost
 remove current from OPEN
 add current to CLOSED

 if current is the target node //path has been found
  return

 foreach neighbour of the current node
  if neighbour is not traversable or neighbour is in CLOSED
   skip to the next neighbour

  if new path to neighbour is shorter OR neighbour is not in OPEN
   set f_cost of neighbour
   set parent of neighbour to current
   if neighbour is not in OPEN
    add neighbour to OPEN
```

Picture 11 Pseudocode

To start with, we create two lists; **OPEN** and **CLOSED**. The **OPEN** list stores all of the nodes for which we have already calculated the F cost. Those are the ones in picture 10 that are marked green. The **CLOSED** list is the set of nodes that have already been evaluated, meaning that those are the ones that are marked red in picture 10. When these two lists have been created, we are adding the starting node to the **OPEN** list. After that, we enter a loop, where we create a temporary variable **current** that is equal to the node in the **OPEN** list with the lowest F cost. That will be the starting node since it is the only one in the **OPEN** list, and we remove the **current** node from the open list, and we add it to the **CLOSED** list. After that, the *if statement* checks if the **current** node is the target node, and it returns the value. That would mean that we found the target node and we can exit the loop. Otherwise, we are going through each of the **neighbouring** nodes of the **current** node. If the **neighbour** is not traversable or it is in the **CLOSED** list, then we just skipping to the next **neighbour** and ignore it. If that is not the case, then we are checking a couple of things if the new path of the **neighbour** is shorter than the old path, or if the **neighbour** is already in the **OPEN** list, we have to update that node to reflect that we found a better path to it. So, if that is true or the **neighbour** is not already in the **OPEN** list, then we set the F cost of the **neighbour** by calculating the G cost and the H cost, and then we set the parent of the **neighbour** to the **current** node. Before the last line, we have to check if the **neighbour** is not in the **OPEN** list. And finally, if that is not true, we are just adding the **neighbour** to the **OPEN** list and we keep looping this. Eventually, in the first *if statement,*

the **current** node is going to be equal to the target node and the path will have been found and we will exit the loop. That is pretty much all there is to it. Once we get into the code, we will look at a couple of things a bit more in-depth. Hopefully, this pseudocode should give you a pretty good understanding of how the A* pathfinding algorithm works.

*dodati screenshot, slijedi: code expl. Za skriptu Node i DrigScript

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

2 references
public class Node
{
    public bool walkable;
    public Vector3 worldPosition;

    0 references
    public Node(bool _walkable, Vector3 _worldPos)
    {
        walkable = _walkable;
        worldPosition = _worldPos;
    }
}
```

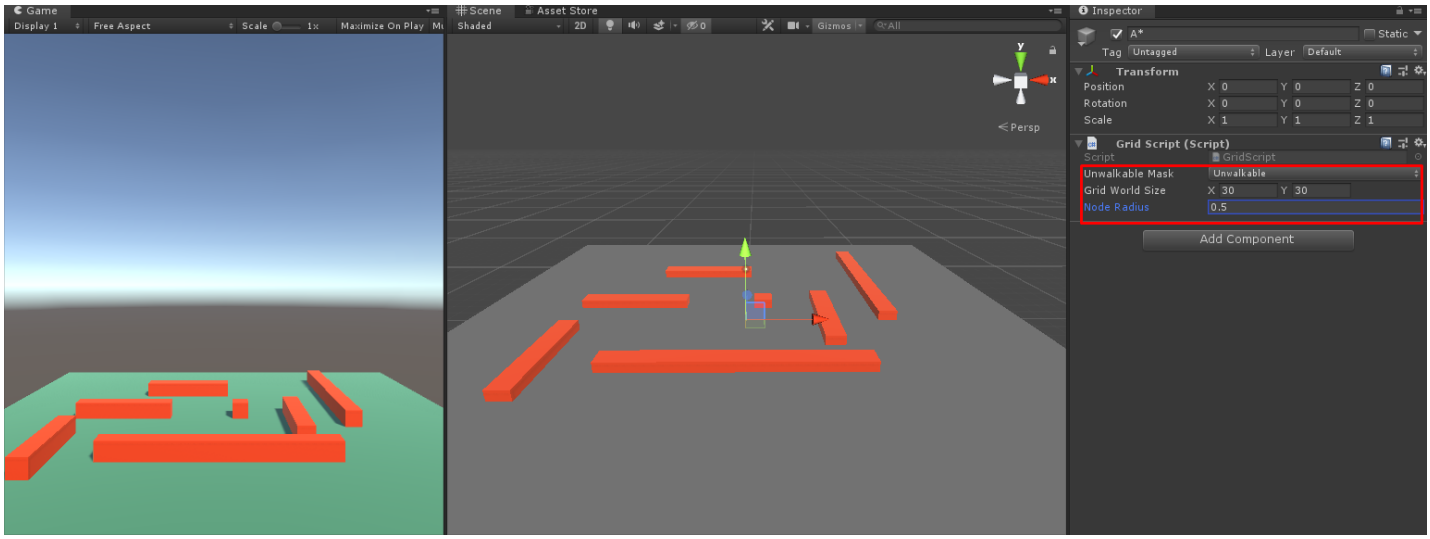
After we create a *Node* script, we can remove the *Monobehaviour* since this is where all the algorithm logic will be placed. We already added layers to the boxes in the Unity, walkable and unwalkable, now we need to create a boolean where the algorithm will know is it walkable or not. After that, we need a public Vector3 where the world position will be stored. Following that, two statements are added to the Node method and *walkable* and *worldPosition.* After that, we're switching to the GridScript script.

```
0 references
public class GridScript : MonoBehaviour
{
    public LayerMask unwalkableMask;
    public Vector2 gridWorldSize;
    public float nodeRadius;
    Node[,] grid;

    0 references
    void OnDrawGizmos()
    {
        Gizmos.DrawCube(transform.position, new Vector3(gridWorldSize.x, 1, gridWorldSize.y));
    }
}
```

First of all, we need a two-dimensional array of nodes called *grid* and a public Vector3 called *gridWorldSize* that will basically define the area in world coordinates that the grid is going to cover. *NodeRadius* will define how much space each individual node covers. Finally, we are adding a public *LayerMask* called *unwalkableMask.*

The *OnDrawGizmos* Unity's method allows us to import objects into the Scene. We'll add *Gizmos.DrawWireCube* with world position and adding new *Vector3* since the *gridWorldPosition* is Vector2. Since we are working in the Unity3D environment and the scene will be played in a 2D spectrum, we must think of the coordinates in a little bit different way. Our grid which will be laid on X and Y coordinates, the Y-axis represents actually the Z-axis in 3D space.

After adding the gizmos, we set the gizmos grid and its values from the inspector.



The X and Y coordinates are the same as the playfield, 30 by 30. Unwalkable layer must be applied to the gizmos and *Node Radius* is set to 0.5 but that can easily be changed later.

Back to *GridScript*, we are adding the *Start* method. First, we need to find out how many nodes can we fit in our grid. For that, we are creating the float *nodeDiameter* and int *gridSizeX* and *gridSizeY above the Start* method.

```
float nodeDiameter;
int gridSizeX, gridSizeY;
0 references
void Start()
{
    nodeDiameter = nodeRadius * 2;
    gridSizeX = Mathf.RoundToInt(gridWorldSize.x / nodeDiameter);
    gridSizeY = Mathf.RoundToInt(gridWorldSize.y / nodeDiameter);
    CreateGrid();
}

1 reference
void CreateGrid()
{

}
```

Since the *nodeDiamater* is a float type, it is necessary to convert it to the integer with the *Mathf* method. After that, we are creating the *CreateGrid* method.

In the *CreateGrid* method, first of all, we want to say that the new *Node* is equal to the new 2D array of nodes, *GridSizeX* and *GridSizeY.* After that, we want to loop through all the positions that the *Nodes* will be in to do a collision check, to see whether they are walkable or not. To make things easier, before the loop and collision checks, outside of the loop, the *worldBottomLeft* vector3 will calculate the bottom left corner of our world.

```
Vector3 worldBottomLeft = transform.position - Vector3.right * gridWorldSize.x/2 -
Vector3.forward * gridWorldSize.y / 2;
```

As X increases, we will go in increments of *nodeDiamater* along with the *world* until we reach the edge. We want the same for the Y-axis (rather Z axis in our world space, as explained earlier). That is why in the code above *worldBottomLeft,* the second parameter is *Vector3.**forward*** rather than ***up.***

```
1 reference
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);
        }
    }
}
```

With this, we are getting each point that the *Node* is going to occupy in our *world.* After that, we want to do a collision check for each of those points. For this, we will obviously need a boolean statement called *walkable.* It will be true if we don't collide with anything in the unwalkable mask. For that, we will use *Physics.CheckSphere* and it returns true if there is a collision. That is why we will use *not true.*

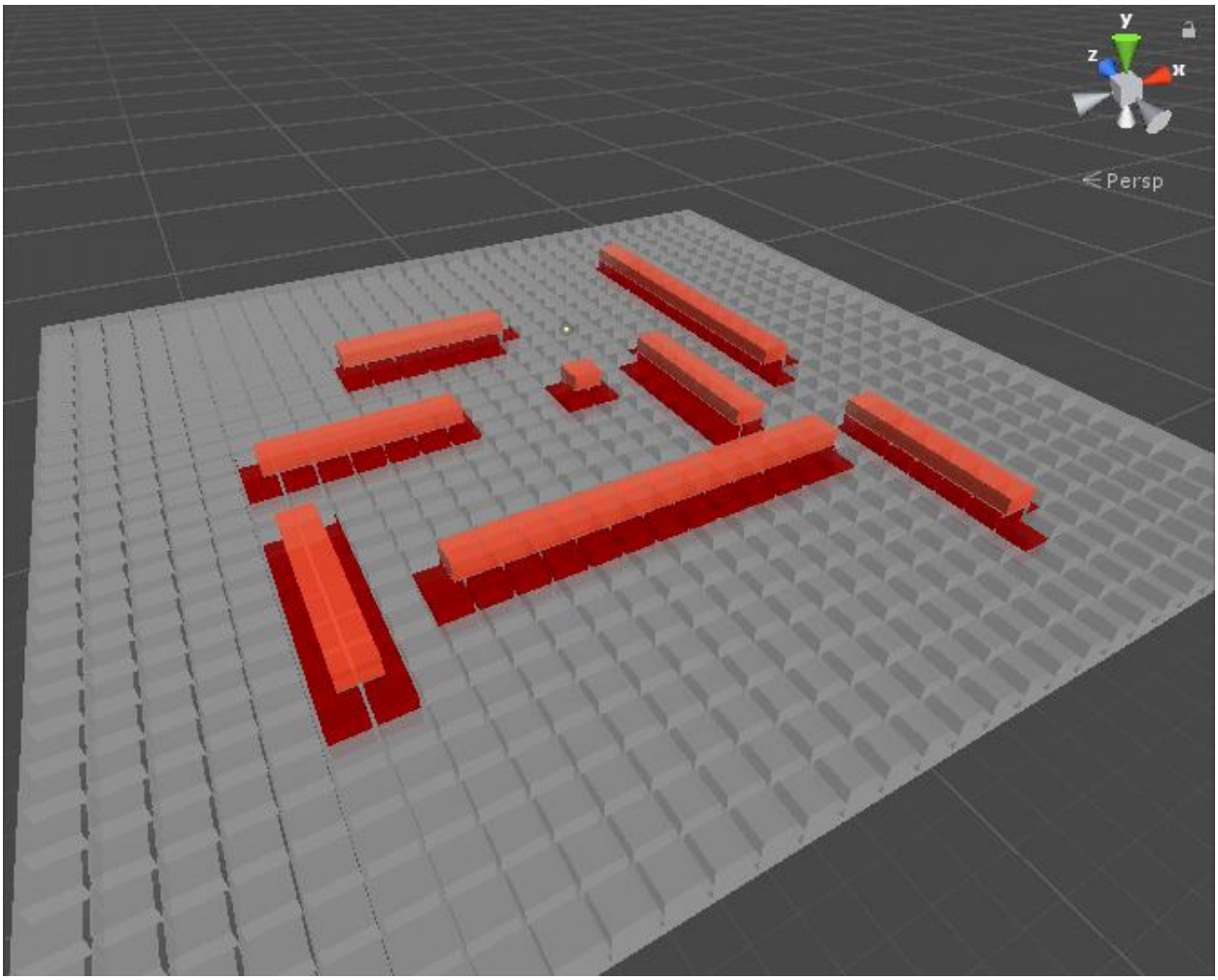
```
for (int x = 0; x < gridSizeX; x++)
{
    for (int y = 0; y < gridSizeY; y++)
    {
        Vector3 worldPoint = worldBottomLeft + Vector3.right * (x * nodeDiameter + nodeRadius) + Vector3.forward * (y * nodeDiameter +
          nodeRadius);
        bool walkable = !(Physics.CheckSphere(worldPoint,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

Now we have to create a new node called *grid* with X and Y-axis that will populate our grid with nodes.

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

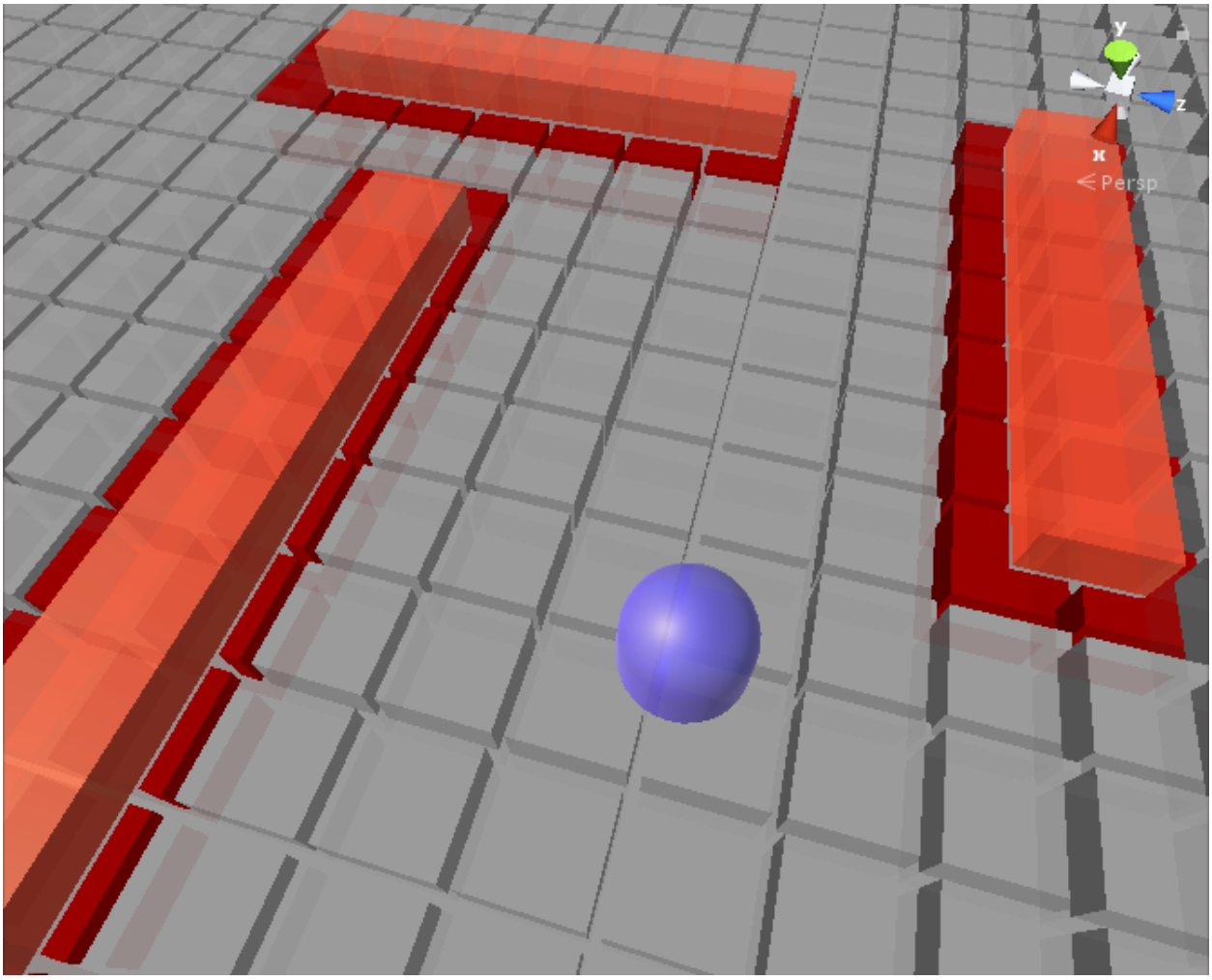
Back into the *Gizmos,* for each node in the grid, we will draw a little cube with *Gizmos.DrawCube*. We need to give it a centre with *n.worldPosition* and the size with *Vector3.one* which is just 1,1,1 in each of the axes and we will multiple it by *nodeDiamater.* To give as some sort of space of outline of each node we will subtract it with -1. Otherwise, everything would be completely filled with red colour. With the subtraction, the grid will be visible.

```
0 references
void OnDrawGizmos()
{
    Gizmos.DrawCube(transform.position, new Vector3(gridWorldSize.x, 1, gridWorldSize.y));
    if (grid != null)
    {
        foreach (Node n in grid)
        {
            Gizmos.color = (n.walkable) ? Color.white : Color.red;
            Gizmos.DrawCube(n.worldPosition, Vector3.one * (nodeDiameter - .1f));
        }
    }
}
```



This is what we got so far. The unwalkable nodes are set to red, the walkable nodes are set to white and the space between boxes is .1f

How do we find the node that is currently standing on the walkable mask?



Let's say we have a player on our grid. How do we find out on which node he is standing on? For that, we need to convert the *worldPosition* into a grid coordinate. We need to create another method that returns a Node called *NodeFromWorldPoint.*

We want to convert this *worldPosition* into a percentage of the X and Y coordinate of how far along the grid it is. So, for the X coordinate, if it is far-left it will have a percentage of 0, if it is in the middle the percentage will be 0.5, and on the far-right, it will have a percentage of 1.

```
float percentX = (worldPosition.x + gridWorldSize.x / 2) / gridWorldSize.x;
```

So, if the *worldPosition* on the X-axis is 0 then we add half the gridworldSize and divide it by the entire *gridWorldSize* and that will give us a half. The same thing we do is for the Y-axis.

```
float percentY = (worldPosition.y + gridWorldSize.y / 2) / gridWorldSize.y;
```

We want these values to be clamped between 0 and 1. We do this with *Mathf.Clamp01*

```
percentX = Mathf.Clamp01(percentX);
percentY = Mathf.Clamp01(percentY);
```

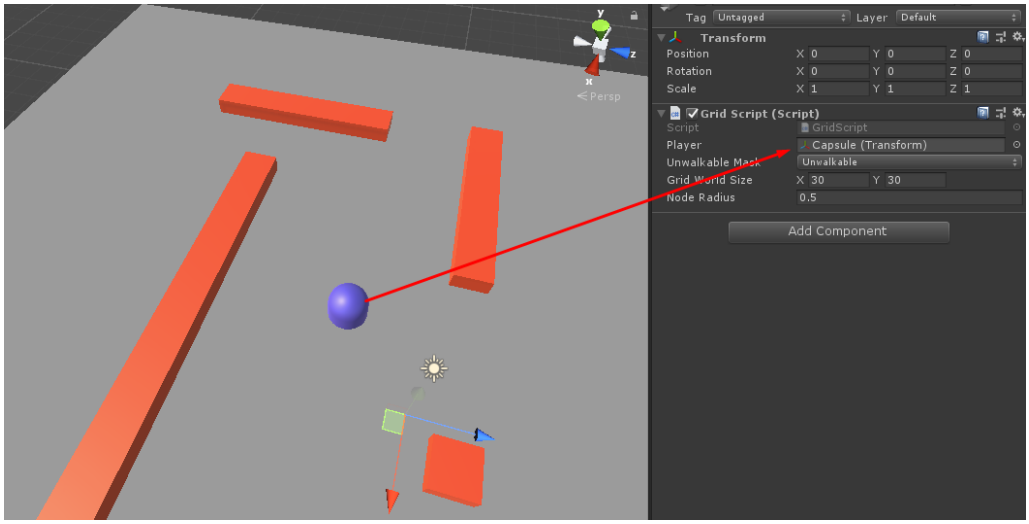
```
public Node NodeFromWorldPoint(Vector3 worldPosition)
{
    float percentX = (worldPosition.x + gridWorldSize.x / 2) / gridWorldSize.x;
    float percentY = (worldPosition.z + gridWorldSize.y / 2) / gridWorldSize.y;
    percentX = Mathf.Clamp01(percentX);
    percentY = Mathf.Clamp01(percentY);

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

Once this is done, let's test it out.

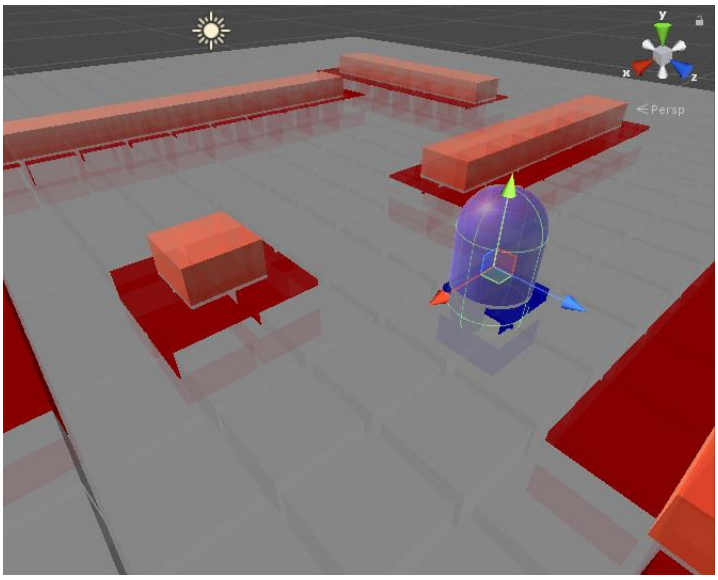
Remember that this player capsule module will be deleted as it will not be part of the project. We will add a capsule to the game and on top of the *GridScript* script, we will make a *public Transform* called player. In the *OnDrawGizmos,* another Node is applied to the *playerNode* and in between *Gizmos.Color* and *Gizmos.DrawCube* a basic *if* statement is set to if *playerNode* is equal to *n* then the *Gizmos.Color* is set to blue.

```
void OnDrawGizmos()
{
    Gizmos.DrawCube(transform.position, new Vector3(gridWorldSize.x, 1, gridWorldSize.y));
    if (grid != null)
    {
        Node playerNode = NodeFromWorldPoint(player.position);
        foreach (Node n in grid)
        {
            Gizmos.color = (n.walkable) ? Color.white : Color.red;
            if(playerNode == n)
            {
                Gizmos.color = Color.blue;
            }
            Gizmos.DrawCube(n.worldPosition, Vector3.one * (nodeDiameter - .1f));
        }
    }
}
```

After finishing this, we are assigning our new placed capsule to the *player* field in the Unity inspector.



Once we added the capsule we can check if everything is working correctly.



You can see that the grid under the capsule is set to be blue. Very nice!

Now we can delete the capsule and remove all of the code since it was just a test to check if everything is working correctly.

Implementing the algorithm

Before writing any code for the algorithm, we have to create a new C# script called *Pathfinding.* This same script we will attach to the A* empty game object created earlier in Unity. The first step that we will do in this script is creating a new method *pathFinding.* It will take two *Vector3*, startPos for the starting position and *targetPos* for the final point on the grid.

The first thing to do is converting these *worldPositions* into *Nodes.* Thankfully, we have already made the method *NodeFromWorldPoint* in our *GridScript* class that is doing just that. We will get a reference of our grid with *GridScript* called *grid.*

```
public class Pathfinding : MonoBehaviour
{
    GridScript grid;
    0 references
    void Awake()
    {
        grid = GetComponent<GridScript>();
        {

        }
    }
    0 references
    void FindPath(Vector3 startPos, Vector3 targetPos)
    {
        Node startNode = grid.NodeFromWorldPoint(startPos);
        Node targetNode = grid.NodeFromWorldPoint(targetPos);
    }
}
```

Now when we prepared everything, this is how it should look like.

```
OPEN //the set of nodes to be evaluated
CLOSED //the set of nodes already evaluated
add the start node to OPEN

loop
  current = node in OPEN with the lowest f_cost
  remove current from OPEN
  add current to CLOSED

  if current is the target node //path has been found
    return

  foreach neighbour of the current node
    if neighbour is not traversable or neighbour is in CLOSED
      skip to the next neighbour

    if new path to neighbour is shorter OR neighbour is not in OPEN
      set f_cost of neighbour
      set parent of neighbour to current
      if neighbour is not in OPEN
        add neighbour to OPEN
```

Back to the pseudocode. We can see that the first thing we have to create is an OPEN set and the CLOSED set. What do we want to do with them? We want to be able to add *Nodes* to them, we want to be able to check if they already contain a specific node and we also need to be able to remove nodes from them.

The OPEN set is a little bit trickier to do because we also want to be able to search it for the node that it contains with the lowest *f-cost.* This is one of the most expensive parts of the algorithm, but we will optimise it later. For now, we will keep it simple.

Back to the code. First what we are doing is making the list of *Nodes* for our OPEN set. The same thing we will do with *HashSet*. From the *pseudocode,* we can see that the next thing to do is adding the starting node to the OPEN set and after that, we are entering the loop.

```
public class Pathfinding : MonoBehaviour
{
    GridScript grid;
    0 references
    void Awake()
    {
        grid = GetComponent<GridScript>();
        {

        }
    }
    0 references
    void FindPath(Vector3 startPos, Vector3 targetPos)
    {
        Node startNode = grid.NodeFromWorldPoint(startPos);
        Node targetNode = grid.NodeFromWorldPoint(targetPos);

        List<Node> openSet = new List<Node>();
        HashSet<Node> closedSet = new HashSet<Node>();
        openSet.Add(startNode);

        while (openSet.Count > 0)
        {

        }
    }
}
```

After repeating the *pseudocode* and preparing everything before writing an actual algorithm, this is what the *Pathfinding* script should contain.

We added the *while* loop that is greater than 0 so it continues looping until it finds the endpoint.

The next thing we need to do is to find the *node* in the OPEN set with the lowest *f-cost.* In order to do that, we have to go back to the *Node* script and add some stuff to it.

The first thing we want to do is creating two public integers called *gCost* and *hCost*. Remember that the *fCost* is equal to *gCost* plus *hCost.*

```
public class Node
{
    public bool walkable;
    public Vector3 worldPosition;

    public int gCost;
    public int hCost;
    1 reference
    public Node(bool _walkable, Vector3 _worldPos)
    {
        walkable = _walkable;
        worldPosition = _worldPos;
    }

    0 references
    public int fCost
    {
        get
        {
            return gCost + hCost;
        }
    }
}
```

This is how the *Node* class looks like now. It is done like this because you will never need to assign the *f-cost* and you will always get it by calculating the *gCost* and the *hCost.* That is why we don't have the *set*, we want to be able just to "get" the *f-cost*.

Back to our *Pathfinding* script. In our *while* loop, we want to loop through all of the OPEN set nodes.

```
while (openSet.Count > 0)
{
    Node currentNode = openSet[0];
    for (int i = 1; i < openSet.Count; i++)
    {
        if (openSet[i].fCost < currentNode.fCost)
        {
            currentNode = openSet[i];
        }
    }
}
```

We set the *i* to be equal to 0 because we set it by default to be 0 in order to continue looping. The *if* statement says that if the OPEN set *f-Cost* is less than the current *Node's f-Cost* then the current node is equal to the open node. As we said, if the two nodes have got an equal *f-cost* then we see which one is

closest to the end node by comparing the *h-cost* and we take the one closest to the end node. That is why we have to put another statement in the *if* statement.

```
while (openSet.Count > 0)
{
    Node currentNode = openSet[0];
    for (int i = 1; i < openSet.Count; i++)
    {
        if (openSet[i].fCost < currentNode.fCost || openSet[i].fCost == currentNode.fCost && openSet[i].hCost < currentNode.hCost)
        {
            currentNode = openSet[i];
        }
    }
}
```

It says: or, if OPEN set's *f-Cost* is equal to current node's *f-cost* and OPEN set's *h-cost* and OPEN set's *h-cost* is less than the current nide's *h-cost* then the current node is equal to the OPEN set [i].

To be clear, this is terribly unoptimized, but we will work on optimization later.

```
OPEN //the set of nodes to be evaluated
CLOSED //the set of nodes already evaluated
add the start node to OPEN

loop
  current = node in OPEN with the lowest f_cost
  remove current from OPEN
  add current to CLOSED

  if current is the target node //path has been found
    return

  foreach neighbour of the current node
    if neighbour is not traversable or neighbour is in CLOSED
      skip to the next neighbour

    if new path to neighbour is shorter OR neighbour is not in OPEN
      set f_cost of neighbour
      set parent of neighbour to current
      if neighbour is not in OPEN
        add neighbour to OPEN
```

Now that we have the node in the OPEN set with the lowest *f-cost* we need to remove it from the OPEN set and add it to the CLOSED set.

```
while (openSet.Count > 0)
{
    Node currentNode = openSet[0];
    for (int i = 1; i < openSet.Count; i++)
    {
        if (openSet[i].fCost < currentNode.fCost ||
        {
            currentNode = openSet[i];
        }
    }
    openSet.Remove(currentNode);
    closedSet.Add(currentNode);
    if (currentNode == targetNode)
    {
        return;
    }
}
```

Simply as that. For now, we will leave this as it is.

Otherwise, we need to loop through each of the **neighbouring** nodes of the current node. In order to do this, we will create another method in the *GridScript* class.

In our *GridScript* class, between *CreateGrid* and *NodeFromWorldPoint* methods, we will create a method called *GetNeighbours.* The first thing we need to know is where exactly is this node in our grid, in our 2D array of nodes. The easiest way to do this is to let the *Node* class keep track of its own position in the array. Back to the *Node* script we will create two public integers, *gridX* and *gridY,* assign them in the constructor with *_gridX* and *_gridY.*

```
public class Node
{
    public bool walkable;
    public Vector3 worldPosition;
    public int gridX;
    public int gridY;

    public int gCost;
    public int hCost;

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

And pass that to the *GridScript* class in our *CreateGrid* method:

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

Now, inside our *GetNeighbour* method, we will create another list of nodes called *neighbours* and set it to the new empty list of nodes. Following that, we have to create a loop that will basically search in 3 by 3 block.

```
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]);
            }
        }
    }
```

If this is relative to the *node's* position, we are searching in 3 by 3 block around the node and when X and Y are equal to 0, then we are in the centre of that block, in that case, we want to skip that iteration because it is not a *neighbouring node,* it is, in fact, the node that was given to us. That is why we must add *if statement* saying if *x* and *y* are equal to 0 and we are skipping this iteration with *continue.*

Otherwise, what we want to check is if this is actually inside of the grid with two additionally added integers *checkX* and *checkY.*

if statement says: if *checkX* is greater or equal to 0 & *checkX* is less than our *gridSizeX* - which is the size of our array, and the same thing for the Y-axis.

Then we are adding this node to the neighbour.

All there is left to do is, outside of the *for loop,* we want to return this list with *return neighbours.*

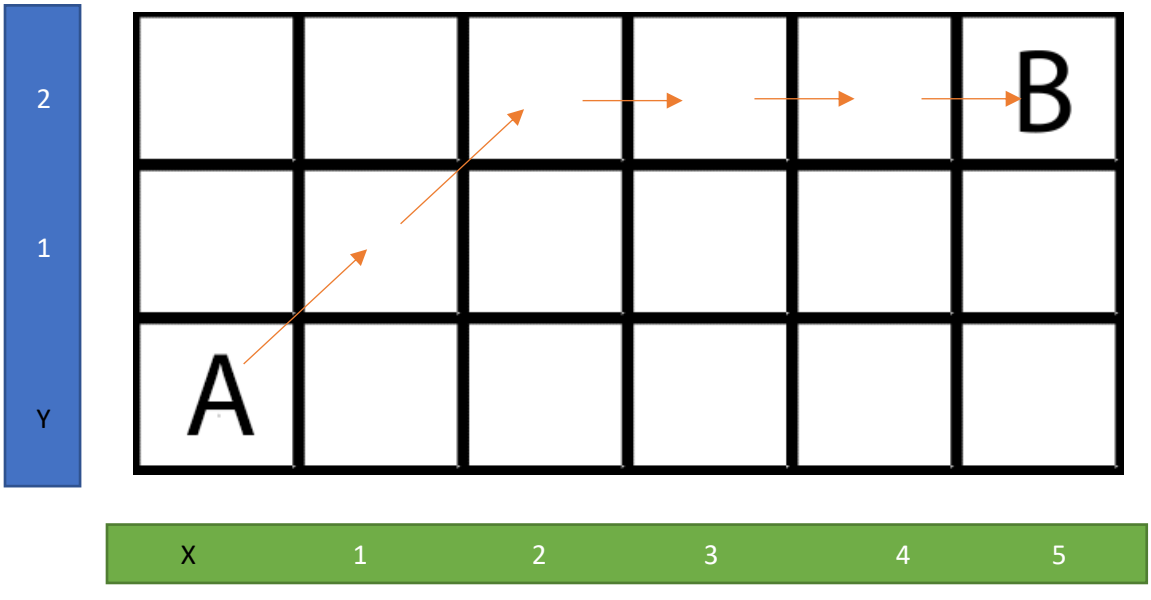
Back to the *Pathfinding* script, we are adding a *foreach* that will loop through all of the neighbours. In the *pseudocode,* we can see that the next step is to check whether the *neighbour* is walkable or not or is it in the CLOSED list - then we want to just skip ahead to the next neighbour.

```
OPEN //the set of nodes to be evaluated
CLOSED //the set of nodes already evaluated
add the start node to OPEN

loop
 current = node in OPEN with the lowest f_cost
 remove current from OPEN
 add current to CLOSED

 if current is the target node //path has been found
  return

 foreach neighbour of the current node
  if neighbour is not traversable or neighbour is in CLOSED
   skip to the next neighbour

  if new path to neighbour is shorter OR neighbour is not in OPEN
   set f_cost of neighbour
   set parent of neighbour to current
   if neighbour is not in OPEN
    add neighbour to OPEN
```

```
while (openSet.Count > 0)
{
    Node currentNode = openSet[0];
    for (int i = 1; i < openSet.Count; i++)
    {
        if (openSet[i].fCost < currentNode.fCost || openSet[i].fCo
        {
            currentNode = openSet[i];
        }
    }
    openSet.Remove(currentNode);
    closedSet.Add(currentNode);
    if (currentNode == targetNode)
    {
        return;
    }

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

Sooner or later, we will be able to get the distance between any two given nodes (point A and point B). For that, we need to make another method in the *Pathfinding* script. The method *GetDistance* will return an integer and with two parameters, nodeA and *nodeB*.

We first count on the X-axis how many nodes away we are from the target node (point B). In this case, we are 5 nodes away from it. On the Y-axis we are 2 nodes away. Now we take the lowest number, in this case, we take the Y-axis with 2 and that will always give us how many diagonal moves will take to be either horizontally or vertically in line with the end node (point B). In this example, we are making two diagonal moves to be in line with the endpoint. Now, to calculate how many either vertical or horizontal moves we need we just subtract the lower number from the higher number. That would be 5 – 2 to give us 3. That is how many horizontal moves we need. The equation would be: $14Y + 10(x-y)$ This is only in the case where the X is greater than Y. If Y was greater than X the equation would be: $14X + 10(y-x)$.



Now when we have a better understanding of it, let's program this. We could have an integer of distance on X-axis named *dstX* and for the distance on the Y-axis integer *dstY*. These integers are equal to the absolute values of the X and Y-axis.

```
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);
}
```

The *if statement* says: if the distance on the X-axis is greater than the distance on the Y-axis then return 14 multiplied by distance Y + 10 multiplied by (distance X – distance Y). Otherwise, we want to do the opposite.

```
OPEN //the set of nodes to be evaluated
CLOSED //the set of nodes already evaluated
add the start node to OPEN

loop
  current = node in OPEN with the lowest f_cost
  remove current from OPEN
  add current to CLOSED

  if current is the target node //path has been found
    return

  foreach neighbour of the current node
    if neighbour is not traversable or neighbour is in CLOSED
      skip to the next neighbour

    if new path to neighbour is shorter OR neighbour is not in OPEN
      set f_cost of neighbour
      set parent of neighbour to current
      if neighbour is not in OPEN
        add neighbour to OPEN
```

What we need to do now is to check if the new path to the neighbour is shorter than the old path or if the neighbour is not in the OPEN list.

Back to our *Pathfinding* script.

```
openSet.Remove(currentNode);
closedSet.Add(currentNode);
if (currentNode == targetNode)
{
    return;
}

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

    int newMovementCostToNeighbour = currentNode.gCost + GetDistance(currentNode, neighbour);
    if (newMovementCostToNeighbour < neighbour.gCost || !openSet.Contains(neighbour))
    {
        neighbour.gCost = newMovementCostToNeighbour;
        neighbour.gCost = GetDistance(neighbour, targetNode);
        neighbour.parent = currentNode;

        if (!openSet.Contains(neighbour))
        {
            openSet.Add(neighbour);
        }
    }
}
```

At this point, the *findpath* method should work. What we need to do now is, once we found the target node, if the current node is equal to the target node, we are exiting the current loop. But, using the *parent,* we first need to retrace our steps to get the path from the start node to the end node.

```
        openSet.Remove(currentNode);
        closedSet.Add(currentNode);
        if (currentNode == targetNode)
        {
            RetracePath(startNode, targetNode);
            return;
        }

        foreach (Node neighbour in grid.GetNeig
    }
}

1 reference
void RetracePath(Node startNode, Node endNode)
{
    List<Node> path = new List<Node>();
    Node currentNode = endNode;

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

Before we exit out of the loop, we call the *RetracePath* method, and it will give the start node and the target node.

Inside the method, we have a list of nodes that is called *path.* A new node is created that is equal to the end node. We will have to trace this path backwards because that is how the parents work.

This is how we will retrace our steps until we reach our starting node at which point, we will have our path.

Currently, the path is reversed, and we have to add the *path.reverse* method.

Testing

We want to be able to visualize our path. For that, we will use *gizmos* once again.

```
    path.Reverse();

    //test
    grid.path = path;
}
```

Inside our *RetracePath* method, we are adding a *grid.path* that is equal to the *path.* Also, on top of the script, we want to add two public *transform* objects, the *seeker,* and the *target.*

```
public Transform seeker, target;
```

After that, we are adding these objects to the newly created *Update* function.

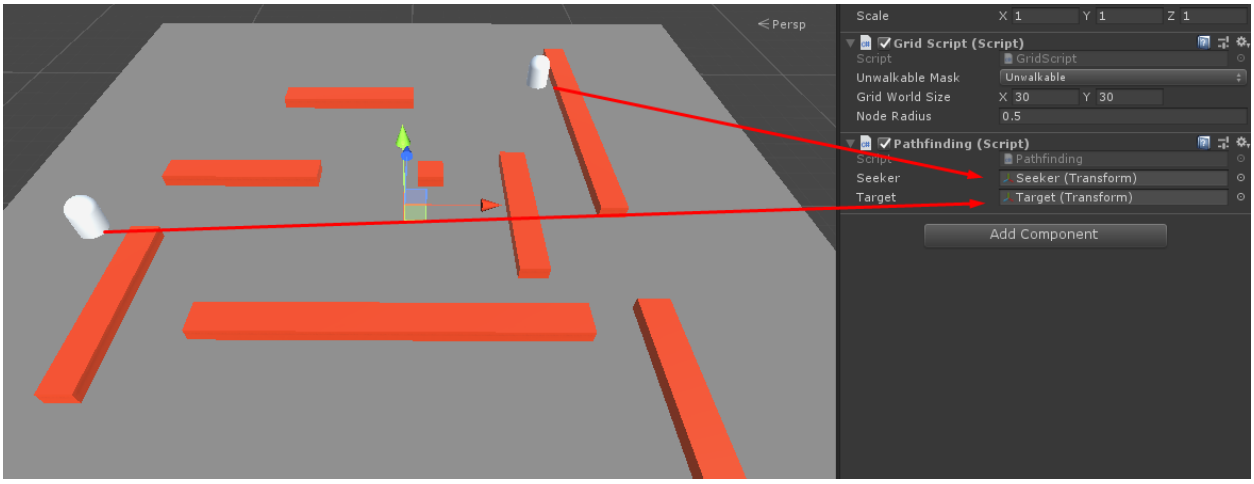
```
FindPath(seeker.position, target.position);
```

```
public List<Node> path;

0 references
void OnDrawGizmos()
{
    Gizmos.DrawCube(transform.position, new Vector3(gridWorldSize.x, 1, gridWorldSize.y));
    if (grid != null)
    {

        foreach (Node n in grid)
        {
            Gizmos.color = (n.walkable) ? Color.white : Color.red;
            if (path != null)
            {
                if (path.Contains(n))
                    Gizmos.color = Color.black;
            }
            Gizmos.DrawCube(n.worldPosition, Vector3.one * (nodeDiameter - .1f));
        }
    }
}
```

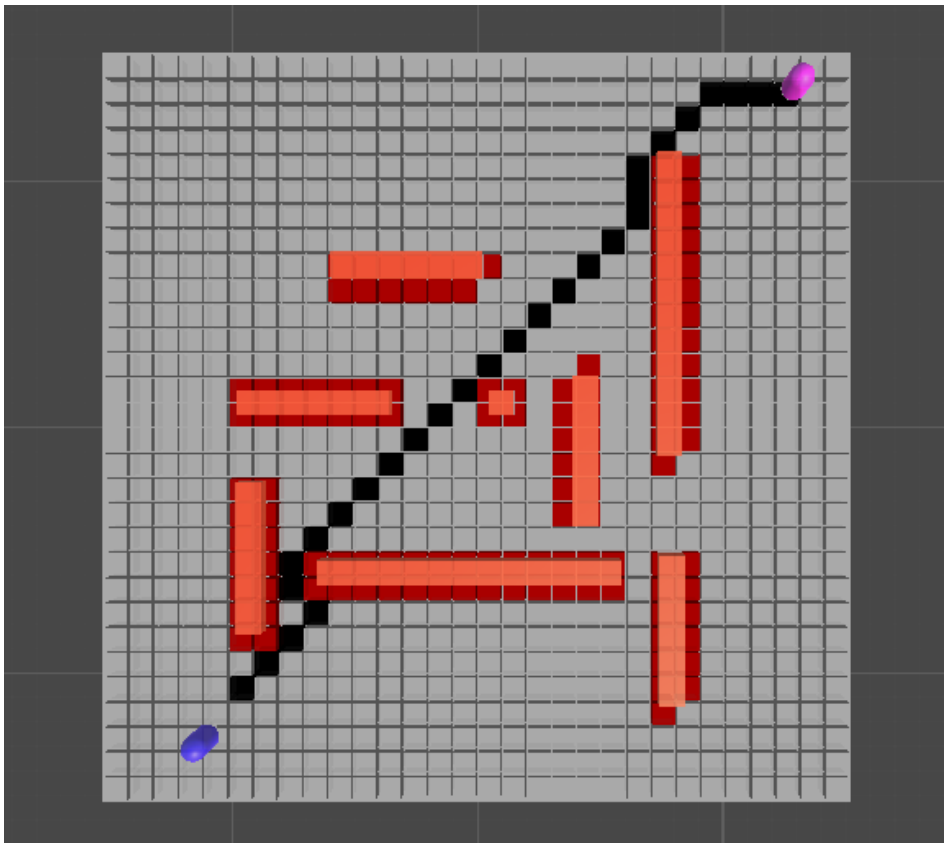
In our *GridScript* class, just above the *OnDrawGizmos* method, we are adding another public list of nodes called *path*.

Right where we are setting up the colors, we are adding: *if the path is not equal to null, then if the path contains the node (n) that we currently looking at, then we can set the gizmos color black.*

Finally, in Unity, we are adding two 3D objects, one called *seeker* and the other one *target*.



Once we save the project, we are ready to run the game. Here is the result:



The pathfinding algorithm works! The next step would be optimization, but I will cover that in the next chapter.