22.9.2021

Oliver Ruane

Freshdumb entertainment

2D Platformer Navigation

Documentation v.0.8

**Table of content**

[Overview 1](#_Toc83339627)

[Basic Concept 1](#_Toc83339628)

[Generating Nodes 2](#_Toc83339629)

[Generating Edges 4](#_Toc83339630)

[Generating Path 6](#_Toc83339631)

[Classes and Methods 7](#_Toc83339632)

[Class NavNode2D : ScriptableObject 7](#_Toc83339633)

[Description 7](#_Toc83339634)

[Fields 7](#_Toc83339635)

[Methods 7](#_Toc83339636)

[GetNeighbour 7](#_Toc83339637)

[Class Edge 8](#_Toc83339638)

[Description 8](#_Toc83339639)

[Fields 8](#_Toc83339640)

[Methods 8](#_Toc83339641)

[Edge (+overloads) 8](#_Toc83339642)

[Class NavMeshManager : MonoBehaviour 9](#_Toc83339643)

[Description 9](#_Toc83339644)

[Fields 9](#_Toc83339645)

[Methods 10](#_Toc83339646)

[RebuildNavmesh 10](#_Toc83339647)

[CreateNodePositions 10](#_Toc83339648)

[AddNodesAlongLine 10](#_Toc83339649)

[TryAddingNodeToList 11](#_Toc83339650)

[ClearDoubleNodes 11](#_Toc83339651)

[CreateGraphNodesFromPositions 11](#_Toc83339652)

[ConstructEdges 11](#_Toc83339653)

[TryAddingEdge 12](#_Toc83339654)

[CheckAllNodesForJumpEdges 12](#_Toc83339655)

[CanReachNode 12](#_Toc83339656)

[SetNodeNeighbour 12](#_Toc83339657)

[EvaluateJumpEdges 13](#_Toc83339658)

[EvaluateSingleJumpEdge 13](#_Toc83339659)

[GetClosestNodeToPosition 13](#_Toc83339660)

[CollisionCheckJump 13](#_Toc83339661)

[Class NavAgent : MonoBehaviour 14](#_Toc83339662)

[Description 14](#_Toc83339663)

[Fields 14](#_Toc83339664)

[Methods 15](#_Toc83339665)

[Update 15](#_Toc83339666)

[SetHeadingNodeAndAdjustPosition 15](#_Toc83339667)

[GetCurrentBufferedTime 15](#_Toc83339668)

[SetClosestMouseNode 15](#_Toc83339669)

[MoveGrounded 15](#_Toc83339670)

[MoveInair 16](#_Toc83339671)

[Class NavMeshData : Scriptable Object 17](#_Toc83339672)

[Description 17](#_Toc83339673)

[Fields 17](#_Toc83339674)

[Methods 17](#_Toc83339675)

[InitNavMeshData 17](#_Toc83339676)

[BuildNavData 18](#_Toc83339677)

[Dijkstra 18](#_Toc83339678)

[GetPathBacktracking 18](#_Toc83339679)

[VisualizeJumpPath 18](#_Toc83339680)

[VisualizePath 19](#_Toc83339681)

[Class NavDataContainer\_Row 20](#_Toc83339682)

[Description 20](#_Toc83339683)

[Fields 20](#_Toc83339684)

[Methods 20](#_Toc83339685)

[NavDataContainer\_Row (constructor overload) 20](#_Toc83339686)

[Class NavigationDataContainer 21](#_Toc83339687)

[Description 21](#_Toc83339688)

[Fields 21](#_Toc83339689)

[Methods 21](#_Toc83339690)

[NavigationDataContainer (constructor overload) 21](#_Toc83339691)

[Results 22](#_Toc83339692)

[Shortcomings 22](#_Toc83339693)

[Learnings and Future Outlook 23](#_Toc83339694)

# Overview

This project is a very basic implementation of a 2D pathfinding system for platformers, which could be used in 2D games.  
It uses a similar approach to traditional tile-based Pathfinding, with some notable differences.  
The nodes in the graph are not limited by a grid system and nodes could theoretically be “infinitely” connected by jumps, which was my main challenge when building this system.

In the example Scene, there is an actor navigating to the node closest to the mouse position.

## Basic Concept

The game needs a graph of nodes and then uses Pathfinding algorithms like A\* or Dijkstra to find a path in the graph.

For simplicity sake lets examine the following Image.

GROUND / ENVIRONMENT

Platform 1

Agent

Platform 2

The Dots represent Nodes in the Graph, and the Arrows represent existing Edges. Notice how the player Agent could reach Platform 2 via Platform 1 but not from the Ground.

This very simple concept could be implemented by placing the Nodes and Edges Manually, thus very precisely controlling where and how the Agent could navigate.

This deterministic way of implementing navigation was not attractive to me as it is very simple and thus not very interesting.   
It also creates a lot of work for the level designers and the Agent is not able to navigate “new level” or randomly generated levels.

My endeavour with this project was to autogenerate as much of the basic concept as possible.

### Generating Nodes

Using polygon colliders (which technically includes Boxes) for the environment allows me to cut the colliders into Line, which can then be segmented, with nodes added along the way.

Take the following geometrie for instance and split it into Lines.

Image : Alternating colours red and blue for clarity).

Now “walk” along the lines and place nodes along the way.

Image

Then evaluate every node for its eligibility.  
It **a)** must be grounded and **b)** must not be inside a collider.  
Both of which can be determined with a single raycast, starting slightly above the node going downward a short distance.

*Image 3*: For simplicity’s sake only example nodes are shown.shows this process. Green nodes have been determined viable. Red ones are discarded.  
Node A was discarded because it has no ground below. Node B was discarded because the ray starts inside a collider.

Node A

Node B

Image : For simplicity’s sake only example nodes are shown.  
Notes:

* The ray is much shorter in practice, but again shown here for clarity.
* Second collider added to illustrate b).

Image : Possible example result.

### Generating Edges

Continuing with the example from generating nodes, the next step is to generate edges.  
If a node exists within a certain small radius from another node, there might be an edge which is then checked.  
The edge is determined not viable, when it goes through a collider.

Image : Process of generating Edges.

After this, one basically has internally connected platforms.

Image   
Note: Every blue node can be reached from every blue node and every purple node from every other purple node.

After that every node which can not be reached by walking from a given node is added to a list to be tested for a possible jump.   
Using *Image 6*  
Note: Every blue node can be reached from every blue node and every purple node from every other purple node. as an example this would mean there might be a viable jump edge from every blue node to every purple node, and thus will be tested for a jump.  
Using the jump height and movement speed Edges can quickly be discarded when exceeding height or horizontal distances possible.

Image : Green Jumps determined possible, Red are not possible

After determining which jumps are theoretically possible, the jumps are checked for collisions along the jump path. (see: *Image 8: Collision check of jump edges*)

Collision boxes corresponding to the Agents Hitbox are placed along the jump path in sub steps and if a collider is hit the jump is deemed not possible and discarded.

This process is good enough to get decent results, but in practice is flawed which I will further extrapolate in [Results and future outlook](#_Results_and_future).

Image : Collision check of jump edges

### Generating Path

Using the generated nodes and edges, distances from every node to every node are precomputed using Dijkstra’s Algorithm and the results are stored in an *n\*n matrix*, where *n* is the number of nodes, storing the distance from source to target and the relevant predecessor.

From this dataset one can easily get a path from one node to another by employing a backtracking algorithm, “walking” from the target to the source via the predecessors.

# Classes and Methods

## Class NavNode2D : ScriptableObject

### Description

Basic class storing information of a graph node.  
Needs to be a Scriptable Object, which is explained in TODO LINK .

### Fields

|  |  |  |
| --- | --- | --- |
| Type | Name | Notes |
| int | m\_id | ID of the Node in the Graph |
| Vector2 | m\_position | World position of the node |
| List<Edge> | m\_neighbours | List of all neighbours stored as Edges to the neighbour |

### Methods

#### GetNeighbour

public Edge GetNeighbour(int \_id)

*Parameters*

|  |  |
| --- | --- |
| \_id | ID of the neighbour to get |

*Description*   
Loops through *m\_neighbours* and return the Edge where *m\_destination.m\_id == \_id*.  
Returns *null* if the Node does not have \_id as a neighbour.

## Class Edge

### Description

Basic class storing information of an Edge.  
Weight is the time it takes going from m\_source to m\_destination.

### Fields

|  |  |  |
| --- | --- | --- |
| Type | Name | Notes |
| float | m\_weight | Predecessor to reach this node |
| NavNode2D | m\_source | Source/Start of the Edge |
| NavNode2D | m\_destination | Destination/End of the Edge |
| bool | m\_isJump | True if the Edge is a Jump.  False by default |
| float | m\_jumpYVelocity | The Initial Vertical Velocity when the Edge is a Jump |
| float | m\_jumpXVelocity | The Horizontal Velocity when the Edge is Jump |

### Methods

#### Edge (+overloads)

public Edge( \* )

*Description*   
Constructors taking different parameters to initialize membervariables.

## Class NavMeshManager : MonoBehaviour

### Description

Main Class handling all the calculation of Nodes and Edges.  
Has an Editor Interface Button to initiate Building the NavMesh etc.

### Fields

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Name | Notes | Default | Scope |
| NavMeshData | m\_navMeshDataSORef | Reference to scriptable Object storing the NavMeshData | 0, 0 | public |
| List<Vector2> | m\_allNodes | List of all Positions eligible for Graph Nodes | *null* | public |
| List<Vector2> | m\_doubledNodes | List of all nodes which have been determined to be doubled.  Used for visualization only. | *null* | public |
| Queue<Edge> | m\_checkForJump | Queue of all Edges that might be a viable Jump | *null* | private |
| float | m\_fPointAccuracy | Distance used to create individual Nodes along the Edges of the Colliders  Smaller Values -> More Nodes | *0.1f* | public |
| float | m\_fDoubleNodeDistance | Distance of two nodes to be determined being the “same” node and prompted for deletion | *0.01f* | public |
| float | M\_iJumpTestSubsteps | Number of Collisionchecks along the possible Jumppath | *20* | public |
| float | m\_fJumpHeight | Max Jump Height to be used for calculation of Jump Edges | *3.0f* | public |
| float | m\_fmoveSpeed | Grounded Horizontal Movement Speed | *4.0f* | public |
| float | m\_fjumpHorMoveSpeed | Aerial Horizontal Movement Speed | *4.0f* | public |
| float | m\_fJumpStartVelocity | Jump Velocity needed to reach m\_fJumpHeight in Environment with m\_gravityMagnitude | *-* | private |
| float | m\_fJumpTime | Time to reach m\_fJumpHeight in default Jump | *-* | private |
| float | m\_gravityMagnitude | Gravity Magnitude | *10.0f* | public |

### Methods

#### RebuildNavmesh

public bool RebuildNavMesh()

*Description*This Method is called when clicking on the Rebuild Navmesh Button in the Editor.  
Reninitializes Lists as well as *m\_fJumpStartVelocity* and *m\_fJumptime*, and then calls Methods to rebuild Nodes and Edges.

Return false when *m\_navMeshDataSORef* is not set and thus *null*.

Finally calls *m\_navMeshDataSORef.InitNavMeshData()* and returns true.

#### CreateNodePositions

void CreateNodePositions()

*Description*Finds all 2D Colliders and iterates through them to create possible node Positions.  
Passes the sides of the Colliders to the *AddNodesAlongLine* Method.

#### AddNodesAlongLine

void AddNodesAlongLine(Vector2 \_point1, Vector2 \_point2)

*Parameters*

|  |  |
| --- | --- |
| \_point1 | Starting point of the given line |
| \_point2 | Ending point of the given line |

*Description*“Walks” along the line in steps with m\_fPointDistance as length and adds possible node positions every step.  
The possible positions are passed to the *TryAddingNodeToList* Method.

#### TryAddingNodeToList

bool TryAddingNodeToList(Vector2 \_pointToAdd)

*Parameters*

|  |  |
| --- | --- |
| \_pointToAdd | Position of possible node position to add to list |

*Description*Casts a small ray downwards to see if:  
a) Inside collider  
- or -  
b) No ground below

If both of these conditions are *false* the point is added to *m\_allNodes* and the method returns *true*.

#### ClearDoubleNodes

void ClearDoubleNodes()

*Description*Checks all positions in *m\_allNodes* if there is another position in *m\_allNodes* with distance smaller or equal than *m\_fDoubleNodeDistance*.  
If so, then one of the positions is deleted and the other is added to *m\_doubledNodes*.

#### CreateGraphNodesFromPositions

void CreateGraphNodesFromPositions()

*Description*Deletes all previous Node Scriptable Objects in the *“Assests/BuiltData/Nodes”* folder.  
Then creates *NavNode2D* Scriptable Objects for all the positions in *m\_allNodes*.

#### ConstructEdges

void ConstructEdges()

*Description*Iterate through all possible Edge combinations and passes the relevant Nodes to the *TryAddingEdge* Method.  
If the Edge is eligible then it passes the new Edge to the *SetNodeNeighbour* Method.

Finally all nodes that are not connected to any other node are removed.

#### TryAddingEdge

Edge TryAddingEdge(NavNode2D \_startNode, NavNode2D \_targetNode)

*Parameters*

|  |  |
| --- | --- |
| \_startNode | Source of the Edge |
| \_targetNode | Destination of the Edge |

*Description*Checks of the Distance of *\_startNode* and *\_targetNode* is below a given Value.  
Then checks whether the given Edge would go through a collider and if so, discards it and returns null.

If the Edge is viable it is added to *m\_navMeshDataSORef.m\_GraphEdges\_temp and is returned.*

#### CheckAllNodesForJumpEdges

void CheckAllNodesForJumpEdges()

*Description*Using breadth-first search alle nodes are checked if they can currently reach every other node using *CanReachNode*.  
If two nodes are currently not connected by any path, the non-existent edge is added to be checked for a possible jump

#### CanReachNode

bool CanReachNode(NavNode2D \_start, NavNode2D \_target)

*Parameters*

|  |  |
| --- | --- |
| \_start | Source of the breadth-first search |
| \_target | Destination of the breadth-first search |

*Description*Returns true of the breadth-first search succeeds.

#### SetNodeNeighbour

bool SetNodeNeighbour(Edge \_newEdge)

*Parameters*

|  |  |
| --- | --- |
| \_newEdge | New edge to set Neighbour |

*Description*Checks whether the source of the current Edge already has the Neighbour given in *\_newEdge*.  
If not, then the Destination in *\_newEdge* is set as a new Neighbour and the Method return *true*.

#### EvaluateJumpEdges

void EvaluateJumpEdges()

*Description*Iterates through all Edges in *m\_checkForJump* and passes them to *EvaluateSingleJumpEdge*.

#### EvaluateSingleJumpEdge

bool EvaluateSingleJumpEdge(Edge \_edgeToCheck)

*Parameters*

|  |  |
| --- | --- |
| \_edgeToCheck | Edge to check for Jump |

*Description*This method looks for a possible efficient jump from *\_edgeToCheck.m\_source* to *\_edgeToCheck.m\_destination*.

If a jump is found, the Neighbours of *\_edgeToCheck.m\_source* are set accordingly and the method returns true.

#### GetClosestNodeToPosition

public NavNode2D GetClosestNodeToPosition(Vector2 \_position)

*Parameters*

|  |  |
| --- | --- |
| \_position | Position to get the closest Node to |

*Description*Returns the Node in *m\_navMeshDataSORef.m\_GraphNodes* closest to *\_position*.

#### CollisionCheckJump

bool CollisionCheckJump(Edge \_edgeToCheck, float \_jumpTime,   
 float \_yVelocity, float \_xVelocity)

*Parameters*

|  |  |
| --- | --- |
| \_edgeToCheck | Edge to check for collisions |
| \_jumpTime | Duration of the jump |
| \_yVelocity | Initial Vertical velocity |
| \_xVelocity | Horizontal Velocity |

*Description*Substeps the jump and performs an OverlapArea at the substep position to check if overlapping with the “environment”. If so, the jump is not possible and false is returned.

## Class NavAgent : MonoBehaviour

### Description

Stores basic information of a simple game Tile.

### Fields

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Name | Notes | Scope |
| NavNode2D | m\_currentNode | Current Node the Agent is currently positioned regarding Navigation | public |
| NavNode2D | m\_closestNode | Closest Node to the Mouse Cursor | public |
| int | m\_headingNodeID | ID of the Node the Agent is currently heading towards. (ie. the next node on the current path) | public |
| int | m\_currentNodeID | The ID of the current Node Mainly used for debugging reason to display in inspector | public |
| Edge | m\_currentEdge | The current Edge the Agent is traveling on.  (Ideally an Edge where the source is m\_currentNode and the destination is a node with m\_id == m\_headingNodeID) | private |
| NavMeshManager | m\_navManagerRef | Reference to the NavMeshManager | public |
| Stack<int> | m\_currentPath | The path the Agent is currently traveling on given as a Stack<int> | private |
| float | m\_bufferedTime | Container to store time buffer when actor would overshoot a node in a single frame | private |
| float | m\_currentMoveTime | Current time passed moving on m\_currentEdge | private |
| float | m\_timeToHeading | Time from m\_currentEdge.m\_source to m\_currentEdge.m\_destination  Mainly used for debugging to display in inspector | public |

### Methods

#### Update

void Update()

*Description*   
Set the closest Node to the Mouse Cursor and then finds a path to that node and navigates towards it.

#### SetHeadingNodeAndAdjustPosition

bool SetHeadingNodeAndAdjustPosition(float \_timeBuffer)

*Parameters*

|  |  |
| --- | --- |
| \_timeBuffer | The leftover time when arriving at node |

*Description*   
Called whenever a node was reached.  
Sets the *m\_headingNodeID* to the next ID in the *m\_currentPath* and adjusts the position of the Actor.

Returns *false* when *m\_currentPath == null*, meaning there is no Edge to currently travel on. Otherwise return *true*;

#### GetCurrentBufferedTime

float GetCurrentBufferedTime()

*Description*   
Returns the current buffered time and resets the value to 0.

#### SetClosestMouseNode

void SetClosestMouseNode()

*Description*   
Sets m\_closestNode using the Mouse Screen Position and pasing it to the *m\_navManagerRef.GetClosestNodeToPosition* method.

#### MoveGrounded

void MoveGrounded()

*Description*   
Process moving along *m\_currentEdge* when m\_currentEdge is not a Jump.

#### MoveInair

void MoveInair()

*Description*   
Process moving along *m\_currentEdge* when m\_currentEdge **IS** a Jump.

## Class NavMeshData : Scriptable Object

### Description

Data Container Storing Build Navigation Data since Data in MonoBehaviour is not persistent.  
Precalculated Data makes pathfinding more efficient during runtime.

### Fields

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Name | Notes | Default | Scope |
| List<NavNode2D> | m\_GraphNodes\_temp | List of all Graph Nodes | *null* | public |
| List<Edge> | m\_GraphEdges\_temp | List of all Edges in Graph | *null* | public |
| List<Edge> | m\_JumpEdges\_temp | List of all Edges, which are Jumps in Graph | *null* | public |
| NavNode2D[] | m\_GraphNodes | Array of all Graph Nodes  Used for efficiency | *0.1f* | public |
| NavDataContainer\_Row[] | m\_GraphNavigationData | Array of NavDataContainer\_Row.  This is basically a nxn Matrix of NavigationDataContainers, where n is number of Nodes. | *0.01f* | public |
| float | M\_iJumpTestSubsteps | Number of Collisionchecks along the possible Jumppath | *-* | public |
| float | m\_gravityMagnitude | Gravity Magnitude | *-* | public |

### Methods

#### InitNavMeshData

public void InitNavMeshData()

*Description*

Called from the *NavMeshManager* after building the Graph Nodes etc.  
Builds an array of *NavNode2D* from the List of Nodes and adjusts the IDs to be the same as the index in the array. This is done for efficiency and for the benefit of random access.

#### BuildNavData

void BuildNavData()

*Description*

Loops through all nodes and perform Dijkstra's Algorithm.  
Then stores results (distance / predecessor) in *m\_GraphNavigationData*.

#### Dijkstra

void Dijkstra(int \_sourceID)

*Parameters*

|  |  |
| --- | --- |
| \_sourceID | ID of the source node in graph |

*Description*   
Dijkstra's Algorithm.

Data stored in *m\_GraphNavigationData.*

#### GetPathBacktracking

public Stack<int> GetPathBacktracking(int \_sourceID, int \_targetID)

*Parameters*

|  |  |
| --- | --- |
| \_sourceID | ID of the source node in graph |
| \_targetID | ID of the target node in graph |

*Description*   
Finds and returns the fastest possible path as a *Stack<int>* from source to target by looking through *m\_GraphNavigationData* in reverse.

#### VisualizeJumpPath

void VisualizeJumpPath(Edge \_jumpEdge)

*Parameters*

|  |  |
| --- | --- |
| \_jumpEdge | Edge to visualize |

*Description*Helperfunction to visualize a given jump. Draws a line in between the substeps.

#### VisualizePath

void VisualizePath(NavNode2D \_source, Stack<int> \_path)

*Parameters*

|  |  |
| --- | --- |
| \_source | The source node of the path |
| \_path | The path to visualize given as a stack of node IDs. |

*Description*Helperfunction to visualize a given path.

## Class NavDataContainer\_Row

### Description

Stores basic information of a simple game Tile.

### Fields

|  |  |  |  |
| --- | --- | --- | --- |
| Type | Name | Notes | Scope |
| NavigationDataContainer[] | m\_GraphNodes\_temp | Array of NavDataContainers | public |

### Methods

#### NavDataContainer\_Row (constructor overload)

public NavDataContainer\_Row(int \_size)

*Parameters*

|  |  |
| --- | --- |
| \_size | Size of *m\_GraphNodes\_temp* |

*Description*   
Constructor with the size of the *NavigationDataContainer* Array as Parameter.

## Class NavigationDataContainer

### Description

Stores basic information of a simple game Tile.

### Fields

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Type | Name | Notes | Default | Scope |
| float | m\_distance | Distance to the relevant source node | *float.MaxValue* | public |
| int | m\_predecessorId | Predecessor to reach this node | *-1* | public |
| bool | m\_visited | Stores if the node was already visited used in Dijkstra’s Algorithm | *false* | public |

### Methods

#### NavigationDataContainer (constructor overload)

public NavigationDataContainer(float \_distance, int \_predecessorId)

*Parameters*

|  |  |
| --- | --- |
| \_distance | Init distance to set |
| \_predecessorId | Init predecessor to set |

*Description*   
Constructor to set \_distance and predecessor.

# Results

## Shortcomings

Although the results are somewhat satisfying, the project suffered somewhat from the theoretically large scope.

I deemed certain functionality to be possible, but in the end did not implement it, since I prioritized an extensive documentation at the expense of some functionality and even then, the documentation leaves room for things to be desired.

Currently Edges are not discarded based on angle, meaning very steep Edges are viable, which can look weird depending on the geometry of the level. It would have been nice to add a setting (like *m\_fMaxWalkableAngle*) for this in the *NavMeshManager*.

Image : Steep Edges are not discarded

Another problem is Nodes not being checked for collision similar to what is done for Jumping. Every Node could easily be checked for the Agents hitbox and if the hitbox does not fit, the Nodes is discarded.  
This could fix some edge cases where Nodes are placed in awkward positions (although currently this rarely happens if ever, since Nodes are deleted when no edges connect to it).

Also, there is currently no way of using multiple Actors regarding each other’s positions as well as moving platforms or changing terrains.  
Furthermore NavMeshData is always saved in the same Folder, thus making it impossible currently to have different Scenes with prebuilt NavData.

Lastly determining if a Jump is theoretically possible is a very complex topic depending on what is allowed. Things like circumventing obstacles mid-air by strafing or stopping movement mid-air to land on a certain node are very specific types of jumps, which can’t be generalized easily and are thus hard to autogenerate.   
On the other hand, a lot of jumps are deemed possible with the current implementation, but only a fraction of them is actually used ever in pathing creating a lot of “garbage jumps” in the form of edges, which are never used. Which again could have been streamlined and I have ideas how to do it, but I just did not find the time to do it.

## Learnings and Future Outlook

In general, implementing the Dijkstra with a weird self-made node structure was insightful. One of the biggest problems were Unity’s serialization rules which I discovered to be a bottleneck for my architecture.

Unity does not allow self-referencing classes to be serialized as this could lead to infinitely deep structures.  
This was obviously due to my bad architecture but did not cause problems until I tried precomputing the NavData and storing it. Self-referencing might be a pattern to be avoided in general, but it was hard to take out of my Structure once I noticed the problem and linked lists use a similar architecture so I assumed it would not be a problem.  
I had to compromise to avoid rewriting most of my code and found the solution to be making the NavNode2D inherit from Scriptable Object. Self-referencing and serialization work fine with Scriptable Objects, so the creation and storing of the Nodes got a little bit convoluted but solved my serialization problems. On a positive Note, this increased debuggability since the Nodes could now be inspected in the hierarchy.

The biggest topic in my opinion is determining possible jumps. Creating logic which helps to find efficient jumps is something to be further examined. One Idea I played with in my head was autogenerating high value mid-air transition nodes, to allow for a variety of new “creative” jumps. This would also require a neat way of storing more complex jumps in edges, like jumps where the Actor changes direction mid-air for example.

Concluding I would like to mention, that I briefly investigated using machine learning to create an “intelligent Actor” able to solve platformer puzzles but decided that to be way out of scope and too complex.