MapNav2

MapNav 2
by Leslie Young
http://www.plyoung.com/

Support and more Info:
http://forum.plyoung.com/c/mapnav

Webplayer sample:
http://www.googledrive.com/host/0BwK5YwVy6AHlSmMzcm1mZHVWSnM

Battlemass 2: (a game that uses a simpler version of MapNav 2)
http://plyoung.itch.io/battlemass-2

Introduction

MapNav 2 will help you get started with a game where you need a tile or node based grid. This is useful for strategy games and board games.

It has various functions you will need when working with a grid map in these types of games; like getting the neighbouring tiles, or finding a path from one tile to another.

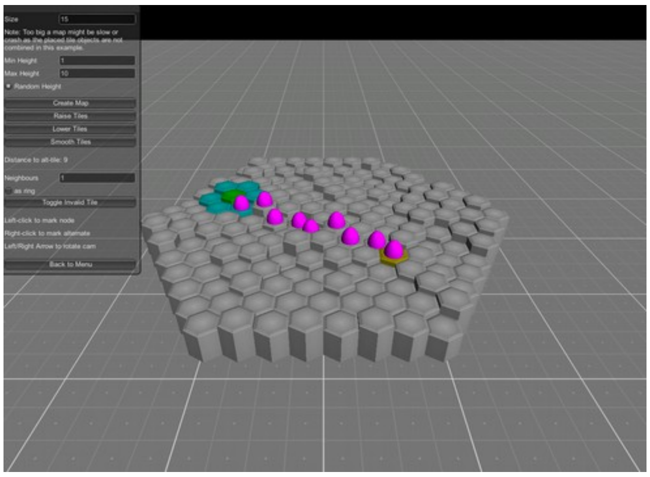
- Full source code (C#)
- Easily create and use custom node/ tile class for properties specific to your game
- All node/ tile data stored in one dimensional array with functions to work with the row and column position of a tile or directly with the tile index into array
- Inspector tools to help with creating and laying out the grid
- Flat-top and Pointy-top Hexagon nodes/ tile layout supported
- Axial and Square (even/ odd-offset) Hexagon node grid supported
- Square nodes/ tiles with 4 or 8-neighbour option
- A* algorithm used to calculate a path
- Various functions needed to work with a grid of nodes, see documentation for more info on what is available in the API

Samples

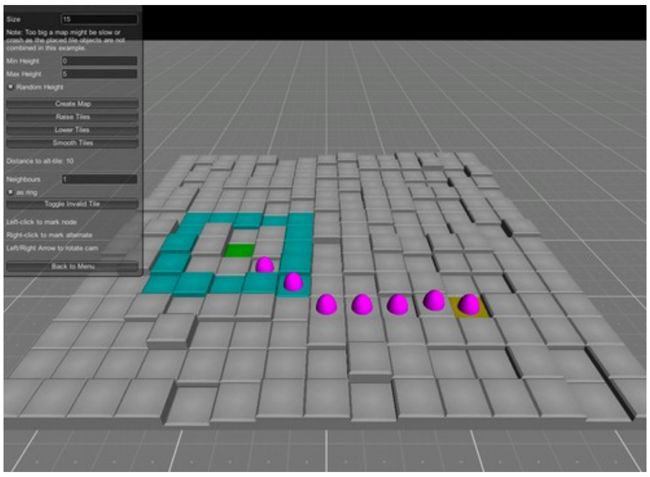
The project includes a few sample scenes, prefabs and scripts to show you how to get started with this kit. These can be found in the `Assets/MapNav2 Sample/` folder.

Open the `Assets/MapNav2 Sample/Scenes/00_menu` scene and press Play to have a look at the included sample scene. Note: you might have to first add all the scenes in that folder to the Build Settings list of scenes (menu: `File > Build Settings)`

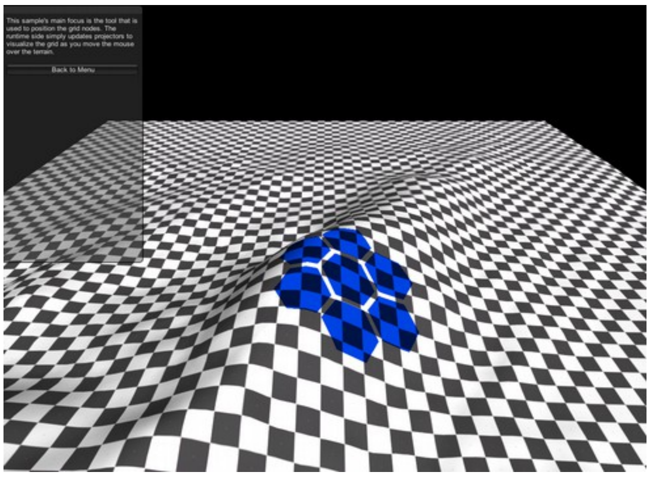
Sample 1 shows you what a hexagon node map in axial layout looks like. The scripts shows how you can create the map at runtime or via editor and how to respond after the grid was generated. It also shows how to find neighbour tiles of a tile clicked on and how to find a path from one tile to another.


Size 15
Note: Too big a map might be slow or crash as the placed tile objects are not combined in this example.
Min Height 1
Max Height 10
Random Height
Distance to alt-tile: 9
Neighbours 1
as ring
Left-click to mark node
Right-click to mark alternate
Left/Right Arrow to rotate cam

Sample 2 is similar to sample 1 but uses a square node map.

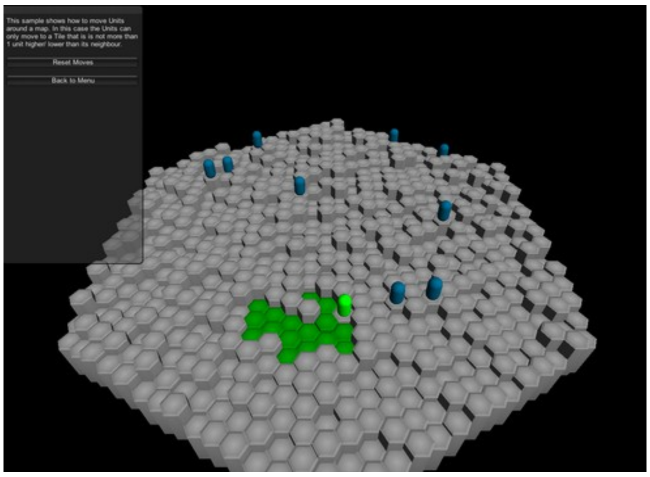

Size 15
Note: Too big a map might be slow or crash as the placed tile objects are not combined in this example.
Min Height 0
Max Height 5
Random Height
Distance to alt-tile: 10
Neighbours 1
as ring
Left-click to mark node
Right-click to mark alternate
Left/Right Arrow to rotate cam

Sample 3 demonstrates how the MapNav Inspector "Height Tool" can be used to create a grid of nodes over a Unity Terrain. The runtime shows how detect over which node the mouse cursor falls.



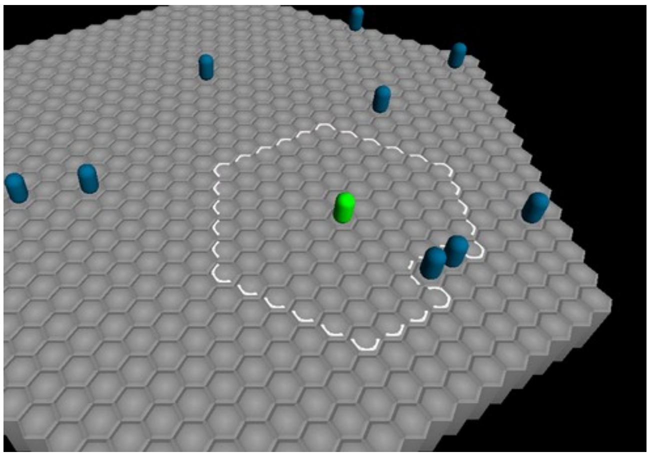
This sample's main focus is the tool that is used to position the grid nodes. The runtime side simply updates projectors to visualize the grid as you move the mouse over the terrain.

Sample 4 shows you how the runtime API can be used to find nodes that a Unit may move to and then calculate a path that the Unit uses to reach a node.



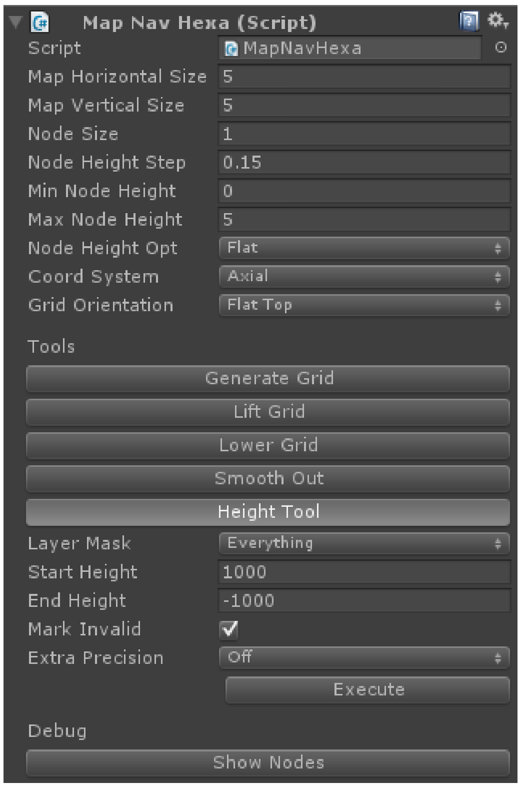
This sample shows how to move Units around a map. In this case the Units can only move to a Tile that is is not more than 1 unit higher/ lower than its neighbour.

Sample 5 extends sample 4 and shows a border around the area that a unit may move rather than marking all the valid nodes.



Sample 6 show how to make use of the NodeLink tool's generated data.

MapNav Properties


Map Nav Hexa (Script)

Script	MapNavHexa
Map Horizontal Size	5
Map Vertical Size	5
Node Size	1
Node Height Step	0.15
Min Node Height	0
Max Node Height	5
Node Height Opt	Flat
Coord System	Axial
Grid Orientation	Flat Top

Tools

Generate Grid

Lift Grid

Lower Grid

Smooth Out

Height Tool

Layer Mask	Everything
Start Height	1000
End Height	-1000
Mark Invalid	✓
Extra Precision	Off

Execute

Debug

Show Nodes

Create a new scene and add an empty GameOject to it. Place either Component `MapNav Hexa` or `MapNav Square` on it, depending on what kind of tiles/ nodes you want in the map.

You will see various options in the Inspector:

Horizontal and Vertical Size is how many tiles you want there to be in the map grid.

Node Size is how big a node is. Tweak this value till it looks right for your situation.

Height Step is used to determine what a Height value in the grid means in terms of world position. If this is set to 0.15 and the node is that H = 0, then in local space (to the grid object) the node is at a Y of 0; for H = 1 that would be Y = 0.15; for H = 2 it is Y = 0.3; etc.

This is used when you use the Lower/ Lift tools of the grid or when gerating a grid with random height values.

When using the "Height Tool" this is ignored since the height for each node is calculated by looking at the point a collider was hit during a ray-cast.

Min/ Max Node Height is the minimum and maximum value the node can be in the grid. This is not the 3D position of the node but the grid position just like the node's row and column position is not its position in 3D/ World space.

Height Opt is the property used to determine how a new grid is generated.

Coord System [Hexa only] determine the shape of the map/ grid. Axial takes on the form of a round/ heagonal map while Odd/ Even-offset is a rectangular map.

Grid Orientation [Hexa only] determines how the hexagon node is rotated.

Diagonal Neighbours [Square only] can be turned on to indicate that the tiles diagonal to another are also considered its neighbouring tiles (8-neighbours total). When it is off then each tile will have only 4 neighbours, the tiles directly above, below and to left and right of any given tile. You will normally choose the one or the other based on how you want to allow units to move 0 either they move diagonally or they may not.

NodeLink Tool

This special tool allows you to add data links between two nodes. A data link is an integer value that is associated with the link between the two nodes.

Click and drag to select the first series of nodes (Blue).
Hold Shift and click-drag to select the second series of nodes (Red).
Moving over a marked node again will deselect it.

You normally create links between neighbouring nodes only but if only one blue and one red node is selected then a data link can be created between the two nodes. If more than one of either is selected then links are created between the blue and red marked nodes that are neighbours of each other.

The value that the link(s) associate with can be accessed at runtime to determine the relation between the two nodes. This can be helpful to determine if there is a wall or door between two nodes. See sample 6 script(s), which shows how this data can be used to handle situation where you have walls and doors betwene nodes.

Integrate

See sample 4 for scripts that does what is explained here.

As is, you can already do a lot with the MapNav (hexa or square) object (grid) and get a lot of information about a node.

In your own game you might however want to add additional information to nodes/ tiles; for example, to track what Unit is occupying the tile, or to keep a reference to some other object in the scene that might be related to the tile.

MapNav makes it easy to extend the basic MapNav node. You simply derive a new class from `MapNavNode` class and then add the additional properties (public variables) to it.

```
using MapNavKit;

public class Sample4Tile : MapNavNode
{
	public Sample4Unit unit = null;
	public string tileName = "";
	public GameObject tileObj = null;
}
```

You will also need to create a custom map class and an inspector for it so that you can inform the "Generate Grid" tool to use your custom node class – this is assuming you are not working in purely runtime code, in which case this is not needed as you will directly call the `MapNavBase.CreateGrid<T>()` function with the type of your custom node/ tile class.

To create a new map class you need to derive from either `MapNavHexa` or `MapNavSquare`, depending on what kind of grid you want.

```
using MapNavKit;

public class Sample4Map : MapNavHexa
{ }
```

The Inspector (editor script) for this new map class needs to be created under a folder or sub-folder of a folder named `Editor`. If you do not do it like this then Unity will not know that the new script is an editor script. You can for example place the script under `Assets/Editor/`

The Inspector class simply needs to derive from either `MapNavHexaInspector` or `MapNavSquareInspector`. Like any Inspector script you need to add the `CustomEditor` attribute and specify which class/ script it is an inspector for. To make the MapNav "Generate Grid" tool us your custom node type you need to add `NodeTpe = typeof(name_of_your_node_class)` in the OnEnable() function.

It will look something like this ...

```
using UnityEngine;
using UnityEditor;
using System.Collections;
using MapNavKit;

[CustomEditor(typeof(Sample4Map))]
public class Sample4MapInspector : MapNavHexaInspector
{
	protected void OnEnable()
	{
		NodeType = typeof(Sample4Tile);
	}

	public override void OnInspectorGUI()
	{
		base.OnInspectorGUI();
	}

	protected override void OnSceneGUI()
	{
		base.OnSceneGUI();
	}
}
```

That is all you need to generate a grid that makes use of your own node type. Now when you use the MapNav API to get a list of nodes you can simply cast the node(s) to the custom type so you can access the properties.

```
Sample4Tile n = map.NodeAtWorldPosition(hit.point) as Sample4Tile;
// or
Sample4Tile n = (Sample4Tile)map.NodeAtWorldPosition(hit.point);

Debug.Log("Clicked on: " + n.tileName);
```

Another benefit of deriving a custom class from the MapNav base classes is that you can override callbacks and respond to events like when a new grid was generated or changed.

You can see this in action in Sample 1 and 2 where I need to know when the grid was created so that I may place the actual tile objects (art) in the scene. Whether you need this function will depend on your game and how you have maps created but it is useful when you need to initialise some data after a map was generated. Have a look at the code of MyHexaMap.cs, MySquareMap.cs, and Sample4Map.cs; at function `public override void OnGridChanged(bool created)` to see how I used this option.

API Reference

Note that is would be best to use the source code as documentation since the property/ function comments will always be more up to date than this reference.

MapNavNode

This is the base class for all nodes/ tiles. Youwil lderive from this when creating a custom node.

public int idx
Index of tile in the grid. Nodes are saved into a one-dimensional array (MapNavBase.grid[]). If -1 then this node is not on the grid. It is possible that some nodes are only "place holders" so that nodes are offset correctly in the array they are stored in. This normally only happens with the Hexa grids when in axial format.

public int q
Position Q (column/x) in 2D grid.

public int r
Position R (row/z) in 2D grid.

public int h
Position H (height/y) used to calculate localPosition using MapNavBase.nodeHeightStep.

public Vector3 localPosition
Position of the node in 3D space, calculated and set when the grid was created/ updated by one of the grid manipulation functions.

public Transform parent
Grid object (MapNav) that owns the node. Used when calculation the position of a node in 3D space. If NULL then this node is not on the grid. It is possible that some nodes are only "place holders" so that nodes are offset correctly in the array they are stored in.

public bool foredInvalid
Use 'isValid' to check if a node is valid. This property, foredInvalid, is used to force a valid node to be invalid. This is mostly used by the tools, like Height Tool, to indicate that a node might be "off" the terrain area but you might use it to make a node invalid for whatever reason. Note that invalid nodes are skipped in just about all functions that processes nodes except those that does something that needs to update this property.

public bool isValid
Returns True if the node is in the grid. If False is returned then this node should be ignored.

public Vector3 position
The position of the node in 3D world space.

MapNav

MapNavBase is the base class from which MapNavHexa and MapNavSquare are derived. You will normally work with either the MapNavHexa or MapNavSquare directly rather than the base class. They both have the same functions and only some properties (variables) differ.

MapNavBase Properties

public delegate bool ValidationCallback(MapNavNode node);
Used when a function needs to check if a node is valid.
- **node**: The node to be checked.
- **returns**: Return True if the node is valid, else False.

public delegate float NodeCostCallback(MapNavNode fromNode, MapNavNode toNode);
Used by the path finding functions. It should return 1 for a normal cost, else it should return 2+ for higher cost and 0 if can't move to the node. The fromNode and toNode are
neighbouring nodes and you may ignore the fromNode if that is not needed for your design.
- **fromNode**: Cost should be calculated for moving from this node.
- **toNode**: to moving to this node.
- **returns**: Return 0 if can't move onto toNode, else 1+ to indicate cost of moving.

public int mapHorizontalSize
The Horizontal size of the grid (along Q/ X).

public int mapVerticalSize
The Vertical size of the grid (along R/ Z).

public float nodeSize
The 3D size of a node in Unity units.

public float nodeHeightStep
The 3D height step of a node. (MapNavNode.y * nodeHeightStep) gives 3D position.

public int minNodeHeight
Min Clamp value for MapNavNode.H (Y)

public int maxNodeHeight
Max Clamp value for MapNavNode.H (Y)

public NodeHeightOpt nodeHeightOpt
How node height will be determined when creating a new grid.
- Flat: Grid is flat, using minNodeHeight.
- Random: Grid nodes have random height between minNodeHeight and maxNodeHeight.

public MapNavNode[] grid
Th grid/ node data

MapNavHexa Properties

public CoordinateSystem coordSystem
How the grid is laid out.

- **OddOffset**: Square shaped grid with off-offset for each node.
- **EvenOffset**: Square shaped grid with even-offset for each node.
- **Axial**: Hexagonal/ roundish shaped grid.

public GridOrientation gridOrientation
Orientation of the nodes in the grid.

- **FlatTop**: The nodes are rotated with the flat side towards the "top" of the grid.
- **PointyTop**: The nodes are rotated with the pointy side towards the "top" of the grid.

MapNavSquare Properties

public bool diagonalNeighbous
Does this grid use the 4 or 8 neighbour system? In a 4-neighbour system the nodes diagonal to another are not considered its neighbours and units can't move directly to them from a specified node.

MapNav Functions

void CreateGrid<T>()
Destroy Grid array and creates a new one from properties.
- T: You can specify a custom Node class derived from MapNavNode

void CreateGrid(System.Type nodeType)
Destroy Grid array and creates a new one from properties.
- nodeType: You can specify a custom Node class derived from MapNavNode

void CreateGrid()
Destroy Grid array and creates a new one from properties.

void ChangeGridHeight(int offs)
Update each node by offset height, clamped to minNodeHeight and maxNodeHeight.

void SmoothOut()
Modify the nodes so that the difference in height between them are not so big.

void AdjustToColliders(LayerMask mask, float startHeight, float endHeight, bool markInvalids, int extraPrecision)
This will update the grid nodes by adjusting the height according to colliders that where hit while casting a ray from start height down to end height. The node's H value will be updated to be an approximation of the height in the grid while the localPosition will be updated to be exactly at the height of where the collider was hit. If no collider was hit along the way then the node's H position will be set to minNodeHeight and the localPosition updated to reflect this position. If markInvalids is set then the nodes that was not over colliders will be set as invalid.
- **mask**: Layer mask to test raycast against.
- **startHeight**: Height to start from.
- **endHeight**: Lowest point to test to.
- **markInvalids**: mark invalid nodes?
- **extraPrecision**: Use extra precision? In this mode not only the center of nodes are checked but also all its corners (4 for square and 6 for hexa). If not all points hit the collider then the node will be considered to not be valid. An average height will be calculated from all the points.
0: Do not use this feature
1: Only check for invalid nodes but use node center to calculate the node height
2: Also update the node height with average of all points.

List<MapNavNode> NodesAround(MapNavNode node, bool includeInvalid, bool includeCentralNode, ValidationCallback callback)
Returns list of nodes starting at the "next" node and going anti-clockwise around the central node.
- **node**: The central node around which to get neighboring nodes.
- **includeInvalid**: Should "invalid" nodes be included? If so then a NULL entry will be added to the returned list for invalid nodes. An invalid node might be one that is stored in the grid array but not considered to be in the grid. This normally happens with Hexa grids. An invalid node might also be one marked as invalid by the callback function.
- **includeCentralNode**: Should the central node be included? It will be added first in the list if so.
- **callback**: An optional callback that can first check if the node is "valid" and return True if so, else it should return False.
- **Returns** null if there was an error.

List<int> NodeIndicesAround(int idx, bool includeInvalid, bool includeCentralNode, ValidationCallback callback)
Returns list of nodes indices starting at the "next" node and going anti-clockwise around the central node.
- **idx**: Index of the central node around which to get neighboring nodes.
- **includeInvalid**: Should "invalid" nodes be included? If so then a -1 entry will be added to the returned list for invalid nodes. An invalid node might be one that is stored in the grid array but not considered to be in the grid. This normally happens with Hexa grids. An invalid node might also be one marked as invalid by the callback function.
- **includeCentralNode**: Should the central node be included? It will be added first in the list if so.
- **callback**: An optional callback that can first check if the node is "valid" and return True if so, else it should return False.
- **Returns** null if there was an error.

List<MapNavNode> NodesAround(MapNavNode node, int range, bool includeCentralNode, ValidationCallback callback)
Return list of nodes in a certain range around given node. The returned list is in no specific order. Excludes invalid nodes.
- **node**: The central node around which to get neighboring nodes.
- **range**: The radius around the central node.
- **includeCentralNode**: Should the central node be included? It will be added first in the list if so.
- **callback**: An optional callback that can first check if the node is "valid" and return True if so, else it should return False.
- **Returns** null if there was an error.

List<int> NodeIndicesAround(int idx, int range, bool includeCentralNode, ValidationCallback callback)
Return list of indices for nodes in a certain range around given node. The returned list is in no specific order. Excludes invalid nodes.
- **idx**: Index of the central node around which to get neighboring nodes.
- **range**: The radius around the central node.
- **includeCentralNode**: Should the central node be included? It will be added first in the list if so.
- **callback**: An optional callback that can first check if the node is "valid" and return True if so, else it should return False.
- **Returns** null if there was an error.

List<MapNavNode> NodesRing(MapNavNode node, int radius, int width, ValidationCallback callback)
Return a list of nodes in a specified range around the given node. The returned list is in no specific order. Excludes invalid nodes.
- **node**: The central node.
- **radius**: Radius from central node that ring starts.
- **width**: Width of the ring.
- **callback**: An optional callback that can first check if the node is "valid" and return True if so, else it should return False.
- **Returns** null if there was an error.

List<int> NodeIndicesRing(int idx, int radius, int width, ValidationCallback callback)
Return a list of node indices in a specified range around the given node. The returned list is in no specific order. Excludes invalid nodes.
- **idx**: The central node index.

- **radius**: Radius from central node that ring starts.
- **width**: Width of the ring.
- **callback**: An optional callback that can first check if the node is "valid" and return True if so, else it should return False.
- **Returns** null if there was an error.

List<int> NodeIndicesAround(int nodeIdx, int radius, NodeCostCallback callback)
This returns a list of nodes around the given node, using the notion of "movement costs" to determine if a node should be included in the returned list or not. The callback can be used to specify the "cost" to reach the specified node, making this useful to select the nodes that a unit might be able to move to.
- **nodeIdx**: The central node's index.
- **radius**: The maximum area around the node to select nodes from.
- **callback**: An optional callback (pass null to not use it) that can be used to indicate the cost of moving from one node to another. By default it will "cost" 1 to move from one node to another. By returning 0 in this callback you can indicate that the target node can't be moved to (for example when the tile is occupied). Return a value higher than one (like 2 or 3) if moving to the target node would cost more and potentially exclude the node from the returned list of nodes (when cost to reach it would be bigger than "radius").
- **Returns** a list of node indices that can be used with grid[]. Returns empty list (not null) if there was an error.

List<MapNavNode> NodesAround(MapNavNode node, int radius, NodeCostCallback callback)
This returns a list of nodes around the given node, using the notion of "movement costs" to determine if a node should be included in the returned list or not. The callback can be used to specify the "cost" to reach the specified node, making this useful to select the nodes that a unit might be able to move to.
- **node**: The central node.
- **radius**: The maximum area around the node to select nodes from.
- **callback**: An optional callback (pass null to not use it) that can be used to indicate the cost of moving from one node to another. By default it will "cost" 1 to move from one node to another. By returning 0 in this callback you can indicate that the target node can't be moved to (for example when the tile is occupied). Return a value higher than one (like 2 or 3) if moving to the target node would cost more and potentially exclude the node from the returned list of nodes (when cost to reach it would be bigger than "radius").
- **Returns** a list of nodes that can be used with grid[]. Returns empty list (not null) if there was an error.

List<int> Path(int startIdx, int endIdx, NodeCostCallback callback)
Returns a list of node indices that represents a path from one node to another. An A* algorithm is used to calculate the path. Return an empty list on error or if the destination node can't be reached.
- **startIdx**: The node where the path should start.
- **endIdx**: The node to reach.
- **callback**: An optional callback that can return an integer value to indicate the cost of moving onto the specified node. This callback should return 1 for normal nodes and 2+ for higher cost to move onto the node and 0 if the node can't be moved onto; for example when the node is occupied.
- **Returns** an empty list on error or if the destination node can't be reached.

List<MapNavNode> Path(MapNavNode start, MapNavNode end, NodeCostCallback callback)
Returns a list of nodes that represents a path from one node to another. An A* algorithm is used to calculate the path. Return an empty list on error or if the destination node can't be reached.
- **start**: The node where the path should start.
- **end**: The node to reach.
- **callback**: An optional callback that can return an integer value to indicate the cost of moving onto the specified node. This callback should return 1 for normal nodes and 2+ for higher cost to move onto the node and 0 if the node can't be moved onto; for example when the node is occupied.
- **Returns** an empty list on error or if the destination node can't be reached.

int Distance(MapNavNode node1, MapNavNode node2)
Returns the distance from node 1 to node 2. Returns 0 on error.

int Distance(int idx1, int idx2)
Returns the distance from node 1 to node 2. Returns 0 on error.

public virtual void OnGridChanged(bool created)
Override this to respond when the grid was changed via functions like CreateGrid and SmoothOut.
- **created**: Will be True if was CreateGrid, else False for all other.

public virtual void OnNodeCreated(MapNavNode node)
Override this for notification while the while a grid is being created. The function is called for each Node being added to the node. You could use this
to add your own initialization data to a custom node being created via CreateGrid[T]()
- **node**: The node that was created.

- eof -