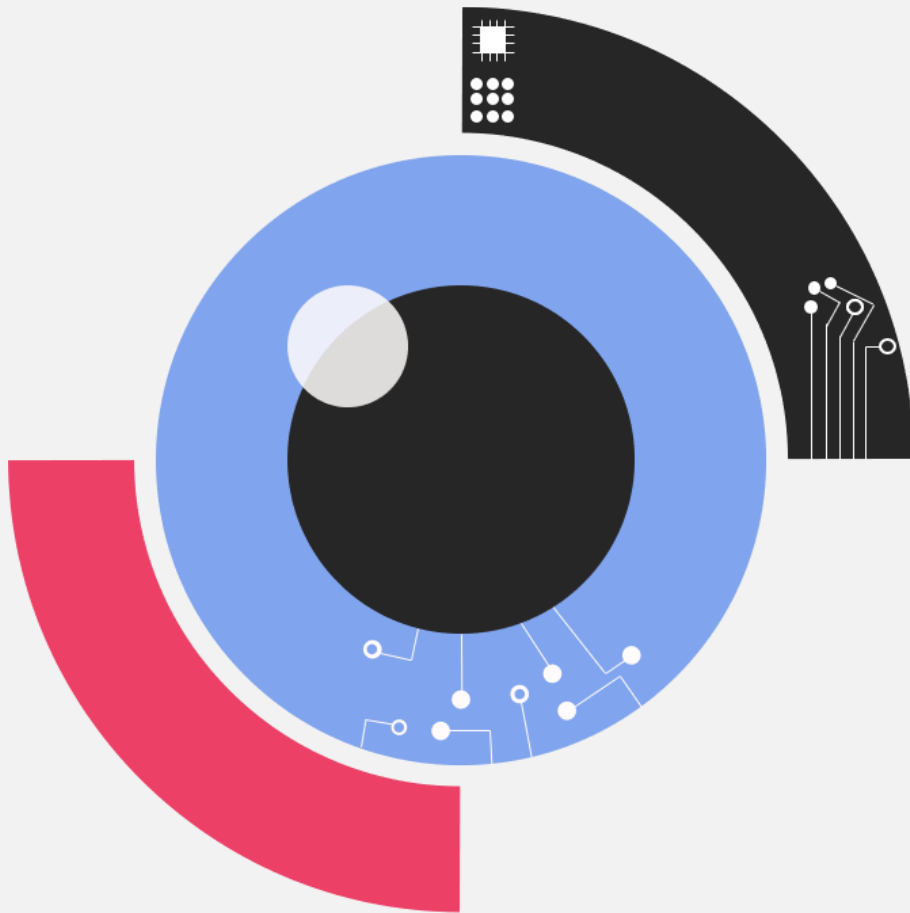# Polymorphic Grid
By
Theory Team

Polymorphic Grid is a collection of classes and functions to create and traverse through a grid.

Of course, we are not talking about traditional square grid only, this toolkit gives support for any shape of grid even if not included in the package.

To get the best of this package you need to understand the whole class hierarchy of Polymorphic Grid.

We are not going to talk about every single variable and method but you can find the deep explanation in the summary of variables, methods and classes inside the scripts.

This file will guide you through all components in the package and give you the ability to use them the best.

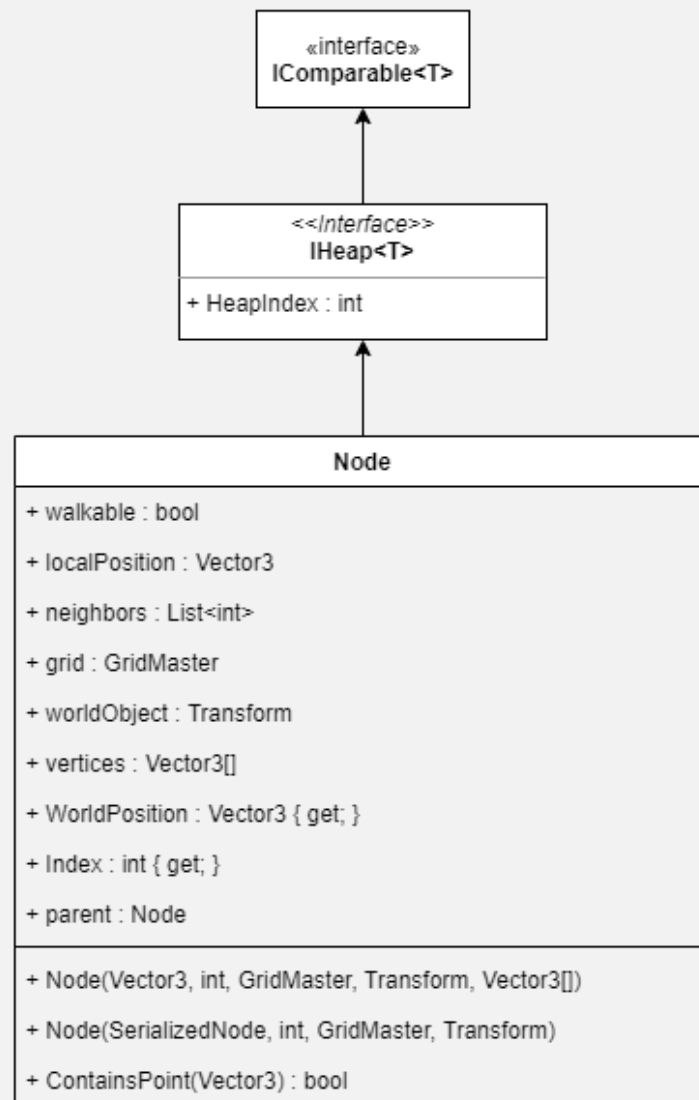You will also find class diagrams for the most parts of the project.

All methods and algorithms are highly optimized, feel free to explore them all.

For any additional information you can read the source code on Github.

Note: To use any of our functionality inside your scripts remember to include 'using TheoryTeam.PolymorphicGrid' at the top of the script.
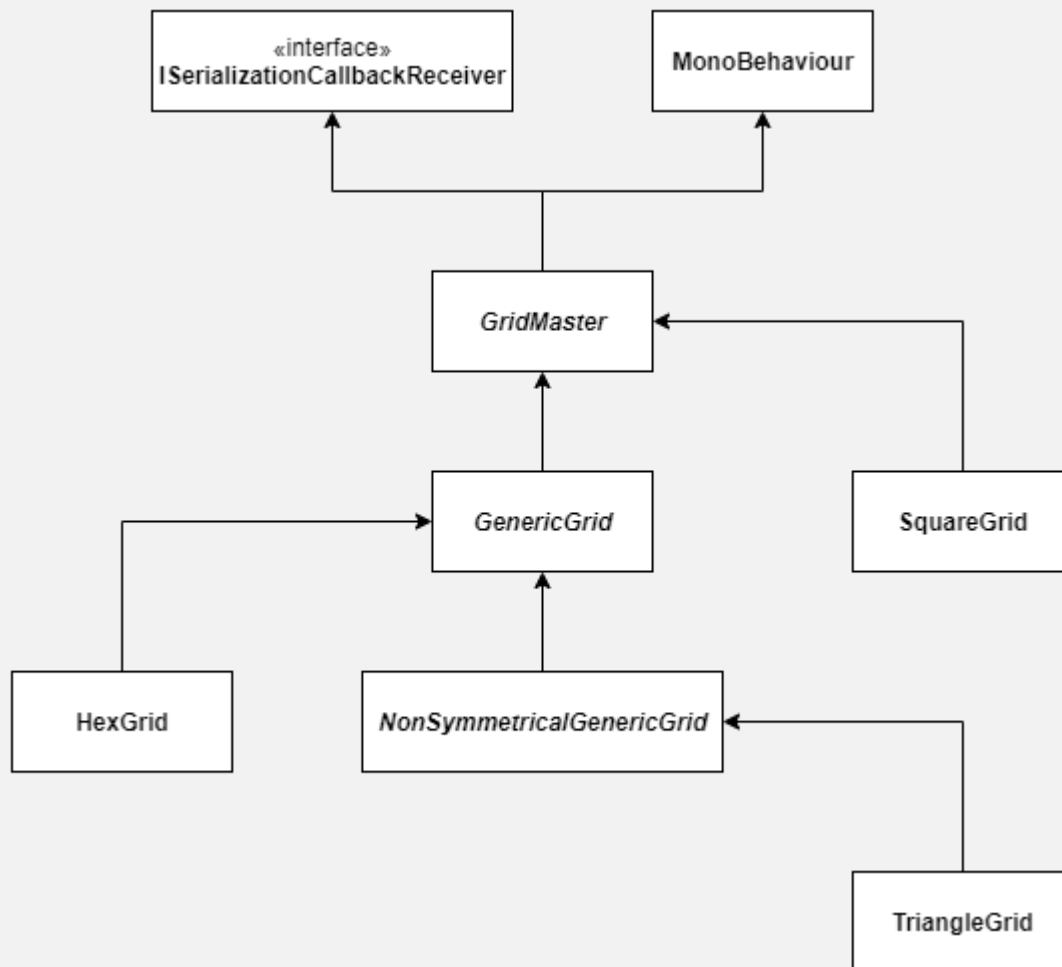
Node

Node is the main class in the package you will find references for it everywhere.

```
              ┌──────────────────────┐
              │     «interface»      │
              │   IComparable<T>     │
              └──────────────────────┘
                         ▲
                         │
              ┌──────────────────────┐
              │    <<Interface>>     │
              │      IHeap<T>        │
              ├──────────────────────┤
              │ + HeapIndex : int    │
              └──────────────────────┘
                         ▲
                         │
```

Node
+ walkable : bool
+ localPosition : Vector3
+ neighbors : List<int>
+ grid : GridMaster
+ worldObject : Transform
+ vertices : Vector3[]
+ WorldPosition : Vector3 { get; }
+ Index : int { get; }
+ parent : Node
+ Node(Vector3, int, GridMaster, Transform, Vector3[])
+ Node(SerializedNode, int, GridMaster, Transform)
+ ContainsPoint(Vector3) : bool

This class represents each node of each grid in the package, it also provides path finding class with needed information and variables to find any path through your grid.

Grid

Now let's talk about the grid itself.

```
   ┌──────────────────────┐         ┌──────────────────┐
   │     «interface»      │         │  MonoBehaviour   │
   │ISerializationCallback│         │                  │
   │      Receiver        │         │                  │
   └──────────────────────┘         └──────────────────┘
              ▲                             ▲
              └──────────┐       ┌──────────┘
                         │       │
                    ┌──────────────┐
                    │  GridMaster  │◄──────────┐
                    └──────────────┘           │
                         ▲                      │
                         │                ┌──────────┐
                    ┌──────────────┐      │SquareGrid│
              ┌────►│ GenericGrid  │      └──────────┘
              │     └──────────────┘
              │          ▲
        ┌─────────┐      │
        │ HexGrid │  ┌──────────────────────────┐
        └─────────┘  │ NonSymmetricalGenericGrid │◄──────┐
                     └──────────────────────────┘        │
                                                    ┌────────────┐
                                                    │TriangleGrid│
                                                    └────────────┘
```

Through this simple hierarchy you can make any kind of grid you wish to create.

If you know exactly how your grid works or even know a better way to make it you should create subclass directly from GridMaster like SquareGrid.

 If you only know the shape of your node in the grid you should create subclass of GenericGrid or NonSymmetricalGenericGrid, both classes will handle the whole functionality for you but the first one is for nodes shape that symmetrical to each other in every place like hexagon, while the other one is for node shapes that needs to be inverted in some cases like triangle.
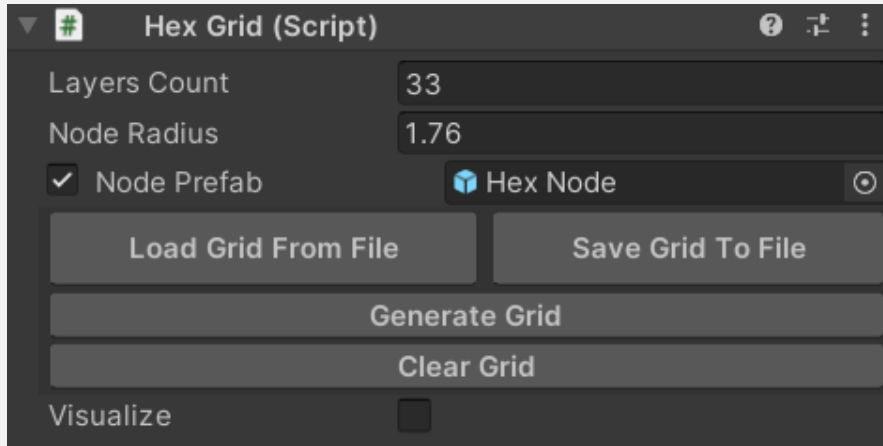
You might think now why not use NonSymmetricalGenericGrid instead of GenericGrid?

Well, you can but it's more efficient to use GenericGrid.

Ok what about using the built-in classes?

As you noticed all grids shares the same concept, they all have GENERATE, CLEAR, SAVE and LOAD buttons in addition to some fields for creating the grid.

You will also notice visualize button to visualize and set walkable and unwalkable nodes in the scene view.

▼ #	Hex Grid (Script)		❷ ⇄ ⁝
Layers Count	33		
Node Radius	1.76		
✔ Node Prefab	🟦 Hex Node		◉
Load Grid From File		Save Grid To File	
Generate Grid			
Clear Grid			
Visualize	☐		
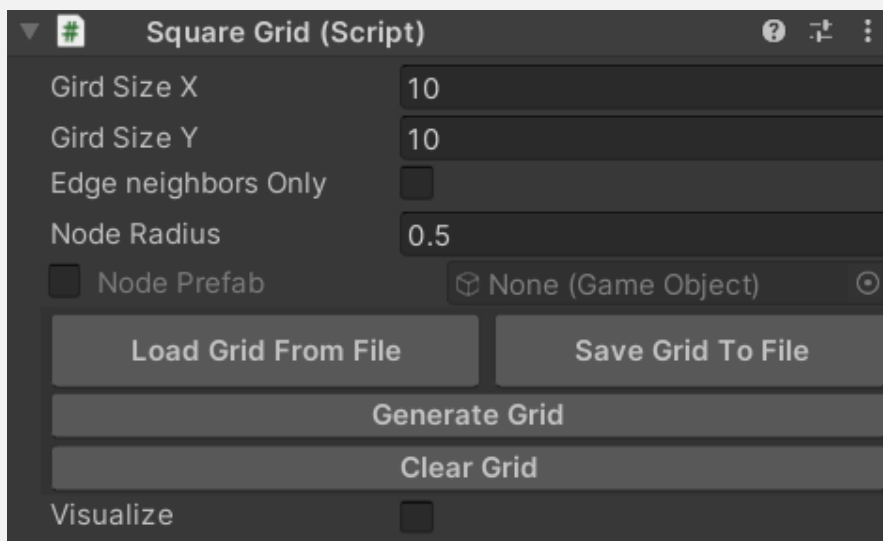
▼ #	Square Grid (Script)		❷ ⇄ ⁝
Gird Size X	10		
Gird Size Y	10		
Edge neighbors Only	☐		
Node Radius	0.5		
☐ Node Prefab	🟦 None (Game Object)		◉
Load Grid From File		Save Grid To File	
Generate Grid			
Clear Grid			
Visualize	☐		
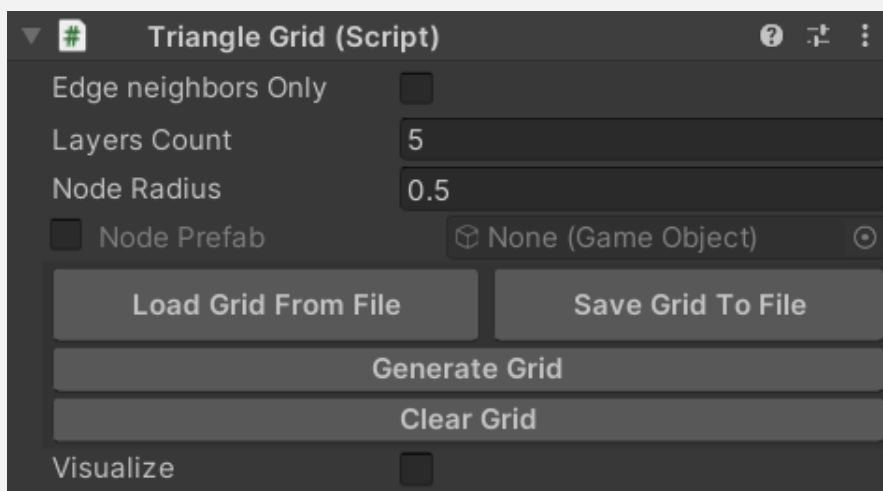
▼ #	Triangle Grid (Script)		❷ ⇄ ⁝
Edge neighbors Only	☐		
Layers Count	5		
Node Radius	0.5		
☐ Node Prefab	🟦 None (Game Object)		◉
Load Grid From File		Save Grid To File	
Generate Grid			
Clear Grid			
Visualize	☐		

Serializable Grid

Generating grid might take a bit of time depending on the size of your grid, so we recommend creating the grid in the editor and then load it on game awake.
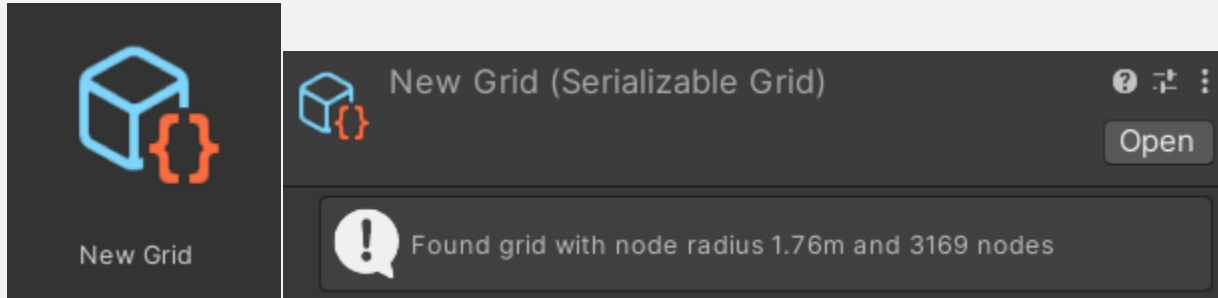

We used scriptable objects to make saving/loading grid easier.

Traversal

It is not specific to grids, it also works on any graph just by defining a couple of methods and variables.

Basically, this class is used to loop through each node in a graph depending on some conditions to apply some functionality.

Of course, all nodes are stored as an array in the GridMaster class but subclasses of Traversal<T> uses graph theory algorithms to traverse through a grid/graph.
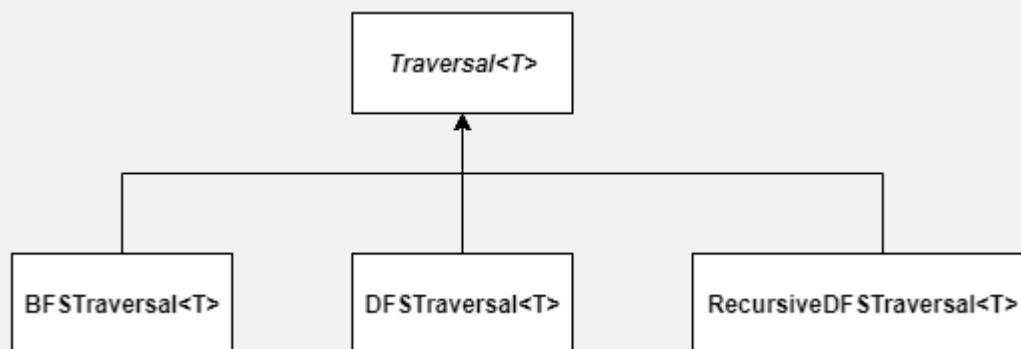
You should be familiar with graph theory and its algorithms to use these classes probably.

You can also write your own traversal by creating subclass of Traversal<T> and implement its abstract method(s).

Note: we do not recommend using Recursive DFS because it has very bad impact on the performance, use BFS Traversal instead.

Grid Visualizer

Visualizing grid in the editor has done by Grid Visualizer this class can also be extended to run during gameplay.
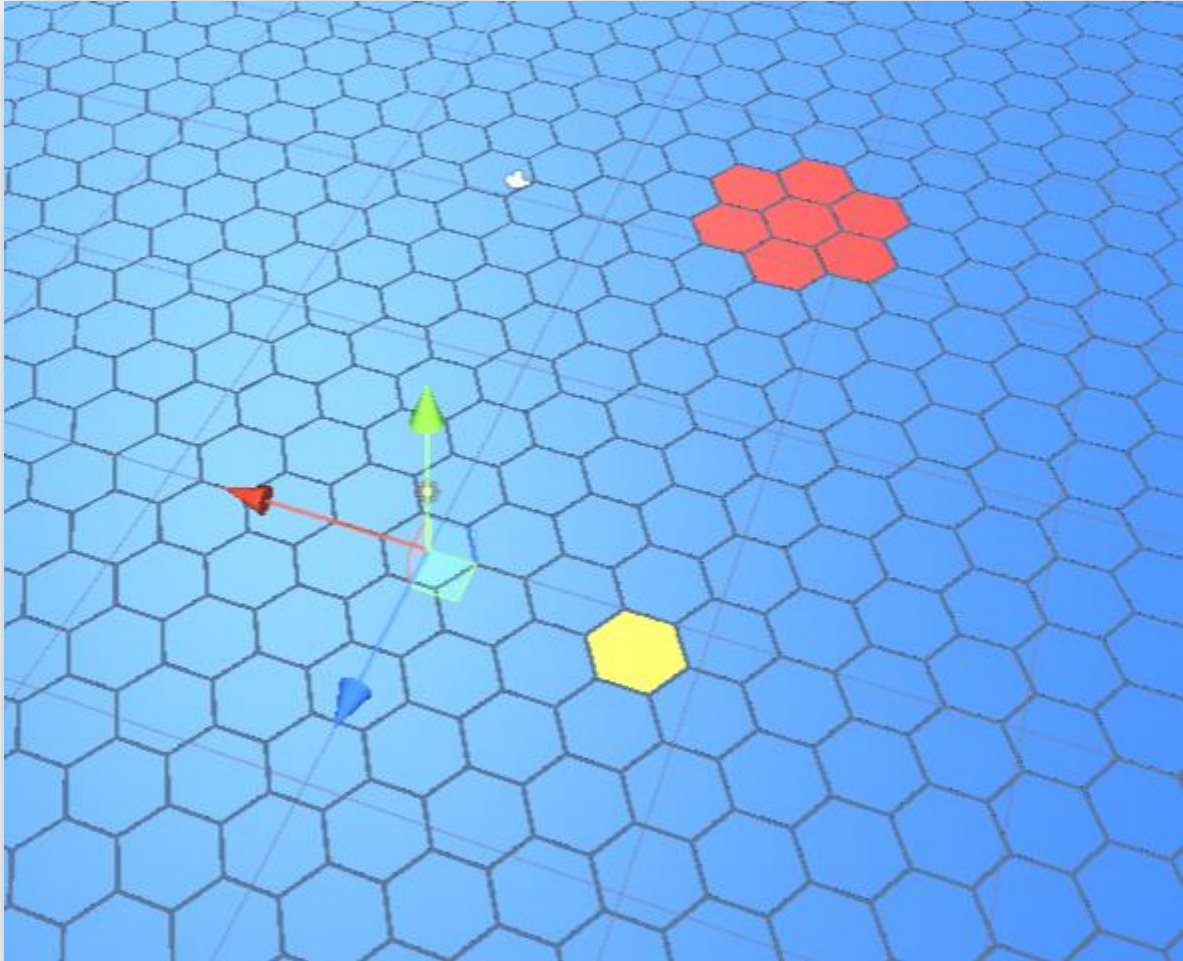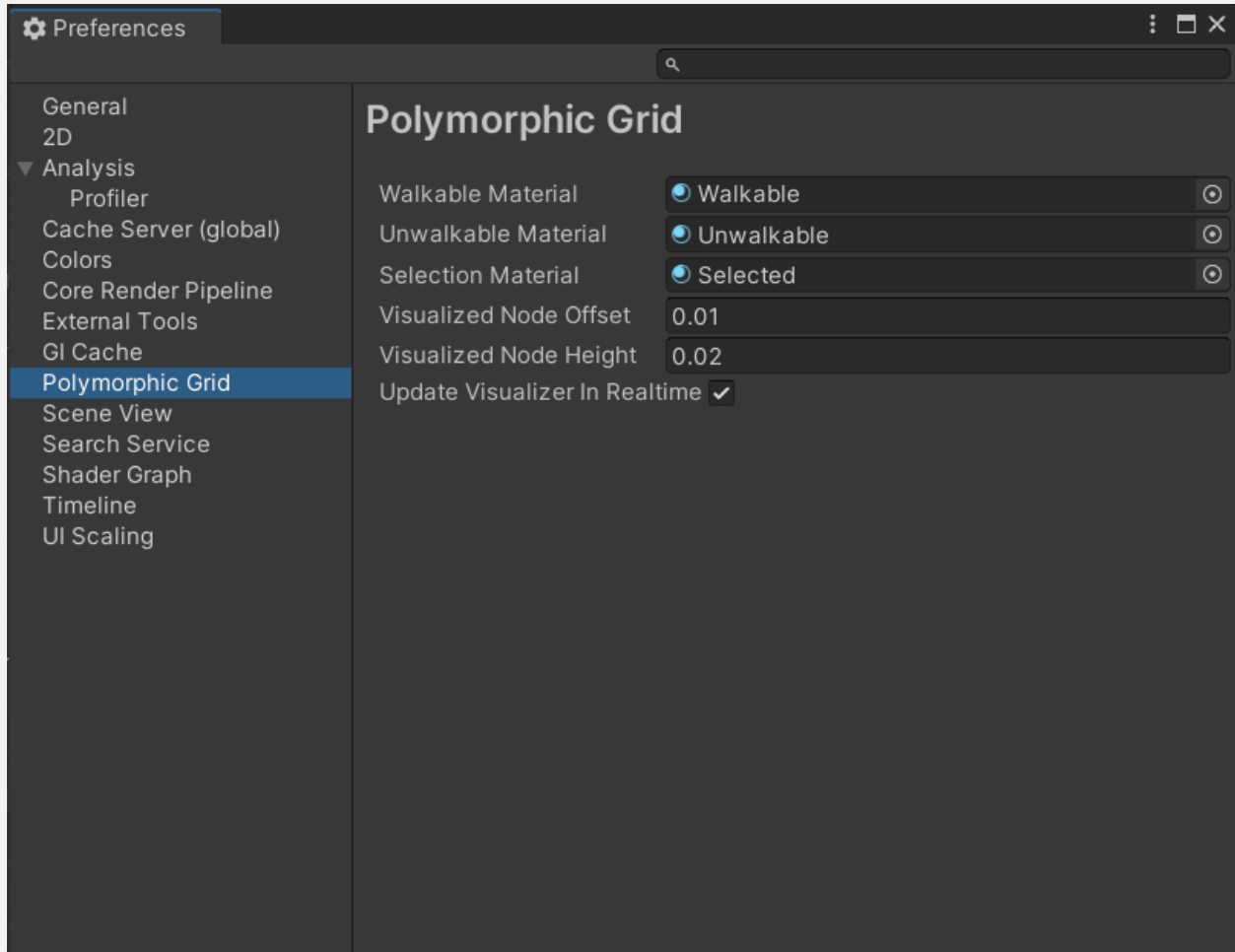

It generates the nodes for you depending on the grid object you assigned.

It also gives you the ability to sort your nodes into groups each group has its own material.

Note: if one or more of the materials are NULL then the all the visualized nodes that hold these materials are going to be deactivated.
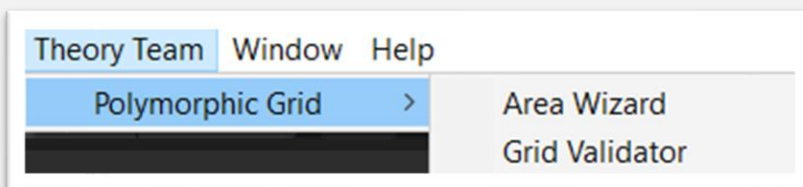
Settings for editor visualizer can be found in Preferences window.

Use Main Settings class to access these settings from your scripts.

It is only available in the unity editor. It won't be included in your build.

There are two more extra tools in the editor we didn't cover yet.

Area Wizard

Although you have the ability to decide walkable/unwalkable areas, you don't need to do it manually.
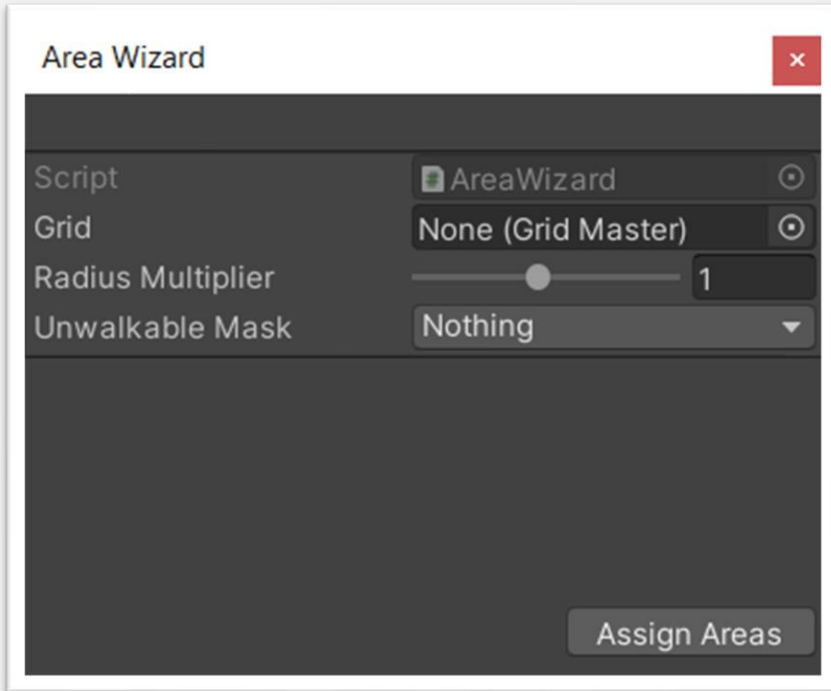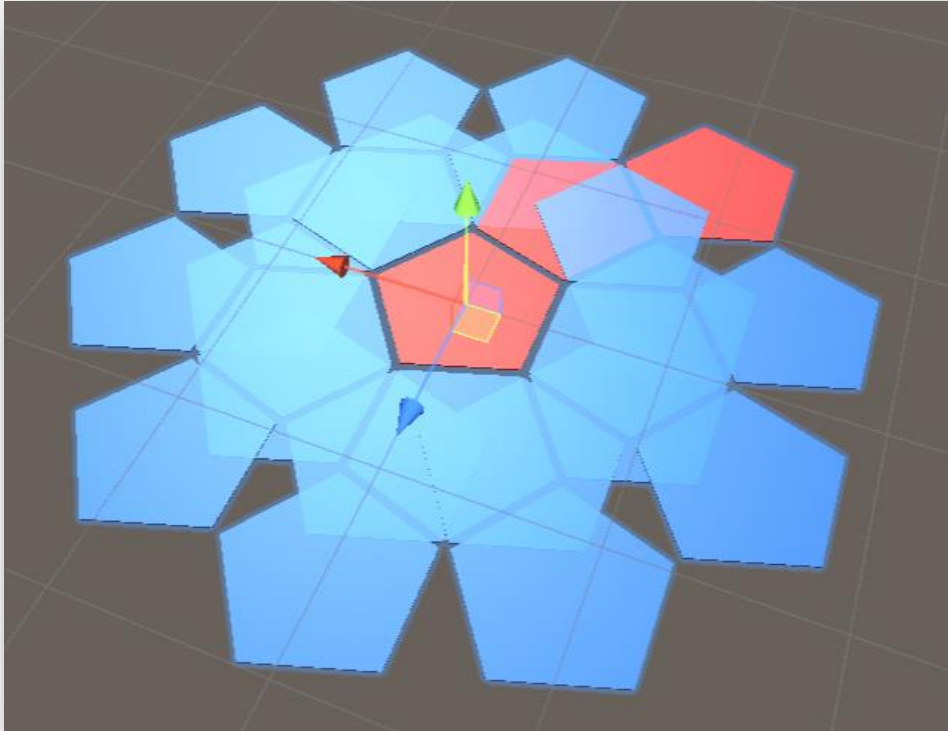

This tool checks each node in a given grid if it collided with something has unwalkable mask to check if is walkable or not.

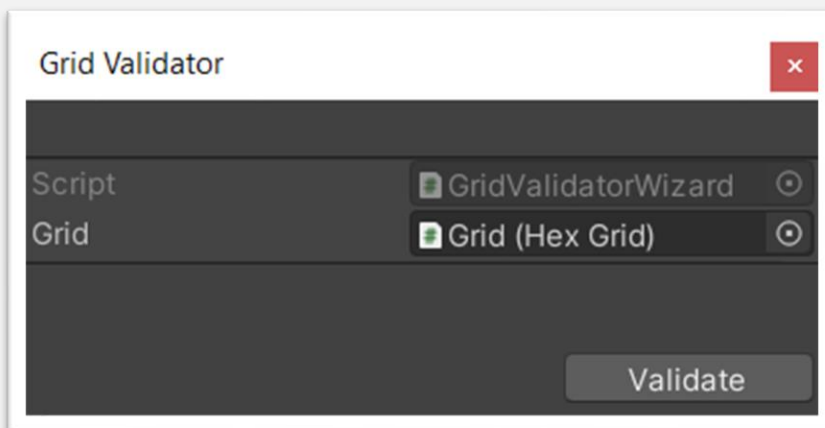Of course, you can adjust walkable and unwalkable areas manually even after using with this tool.

Grid Validator

Unfortunately, not all shapes can be nodes in a grid.

As you noticed, the system does not have a problem with any shape and can deal with it any way.

However, if you want to check weather your grid is valid or not you can use grid validator.

For faster grid validation, we don't recommend using this tool with too many nodes grid. And be sure to generate your grid before validation.

Path Finder

It might look simply but it does a lot of work.
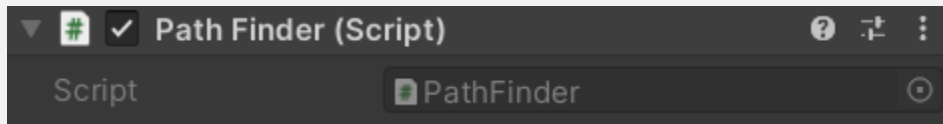
This class perform A Star and BFS algorithms to find paths.

It will work on any grid that inherits GridMaster.

It also can work on a separate thread so you don't have to worry about frame drops.

Static methods work on the main thread, however public methods work on a separate thread.

Data Structures

This package also includes implementations for Heap and Binary Search Tree data structures to speed up all its operations.

Here are the most important methods in this package with their time complexities:

<u>Generating Grid:</u> O(n.log(n)) where n is the number of nodes in the grid.

<u>Converting 3D position into its appropriate node:</u> O(log(n)).

Those are true for hex and triangle grids but square grid is a little bit faster.

<u>Generating Grid:</u> O(n).

<u>Converting 3D position into its appropriate node:</u> O(1) for square grid.

General operations:

<u>Finding a path:</u> O(n).

<u>Traversing through a grid using BFS or DFS:</u> O(n).

<u>Traversing through a grid using recursive DFS:</u> $O(n^m)$ where n is the number of nodes in the grid,

and m is the depth of the traversing.

For hex grid, n = layers count * (layers count – 1) * 3 + 1

For triangle gird, n = layers count * (layers count – 1) / 2 * 3 + 1

For square grid, n = grid size x * grid size y

Finally

If you have any other question, you can always contact us on these emails:

Theoryteam.2000@gmail.com

nitrobally@gmail.com

tarek.nice.2001@gmail.com

Please don't forget to rate the asset and give us your feedback

Thanks for using our asset.