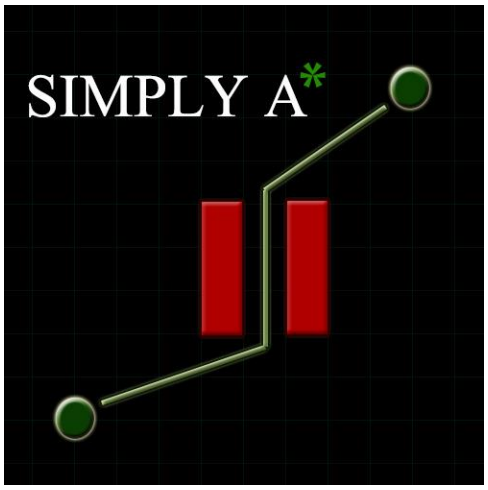
SIMPLY A* - A SIMPLE SOLUTION!

DOCUMENTATION

INTRODUCTION

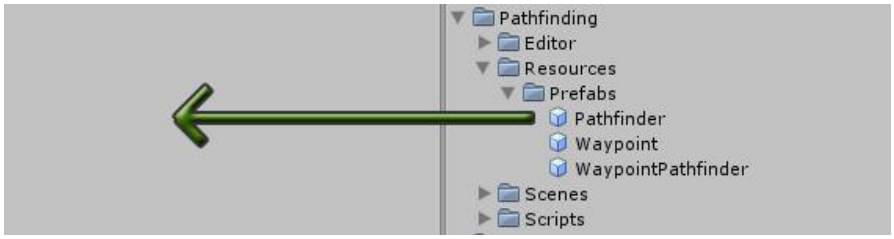
First of all, thank you for showing interest in Simply A*!
Simply A* is an easy to use, fast and simple pathfinding solution. It is divided into two pathfinding approaches: a grid based and a waypoint based approach. The different solutions will be discussed individually but is built on the same foundation. The source code is fully available if tweaks are needed for your project. You can find a Tutorial video at:
http://www.youtube.com/watch?v=ZLrYaLOi1zo&feature=youtu.be



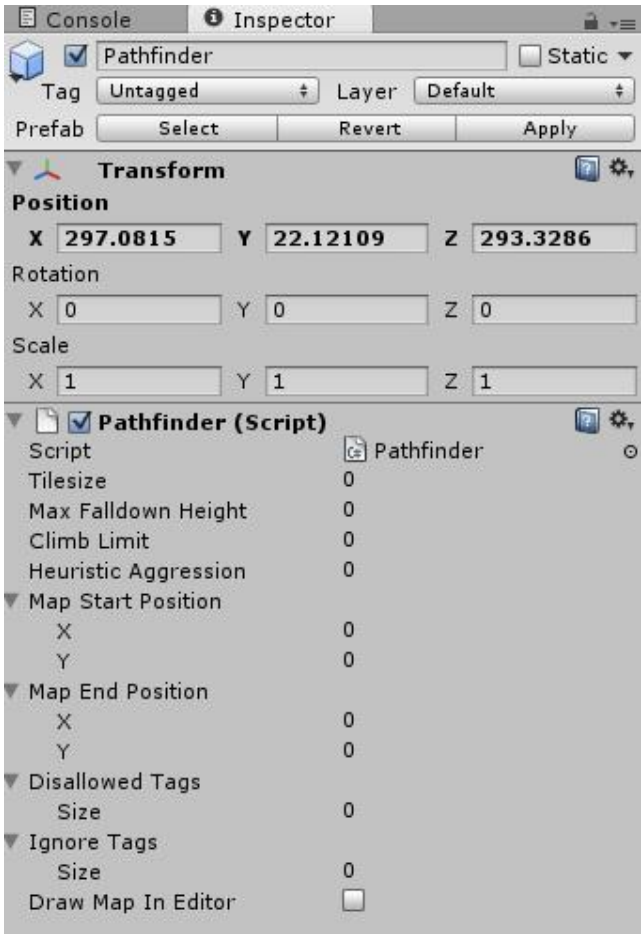
THE GRID BASED SOLUTION

The grid based solution is really easy to setup:

1. Drag the **Pathfinder** Prefab into the scene, it is found at *Pathfinding/Resources/Prefabs.*

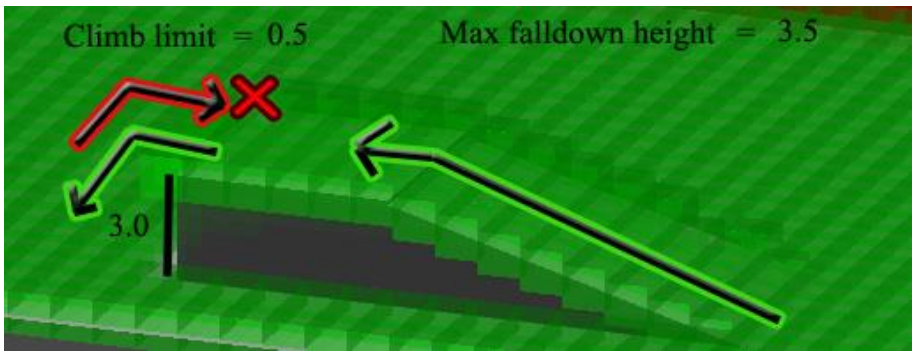


2. Click on **Pathfinder** object in the scene hierarchy and watch it in the **Inspector.**



3. Start by setting the **Tilesize.** It needs to be bigger than zero. The smaller the **Tilesize** is the more refined and detailed the map will be, but it will also be a map with more *nodes* and thereby have heavier performance. I suggest no lower than 1. If you have a simple map use a higher **Tilesize**, if you have a complex map use a lower **Tilesize.**

4. Now we need to set **Climb Limit** and **Max Falldown Height.** Climb limit decides how steep we can climb and max falldown height decides how far we will be able to jump down. If the map is flat then keep both at zero.

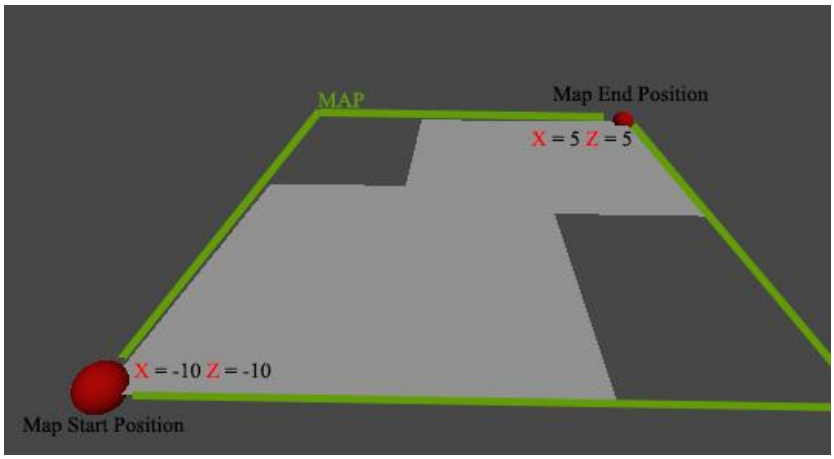


5. Heuristic Aggression is the next we should setup. It should be **ZERO!** In almost any cases however if you have an extremely simple map then it could be boosted above 1. It will become faster performance wise but not always return the best path, just a path.

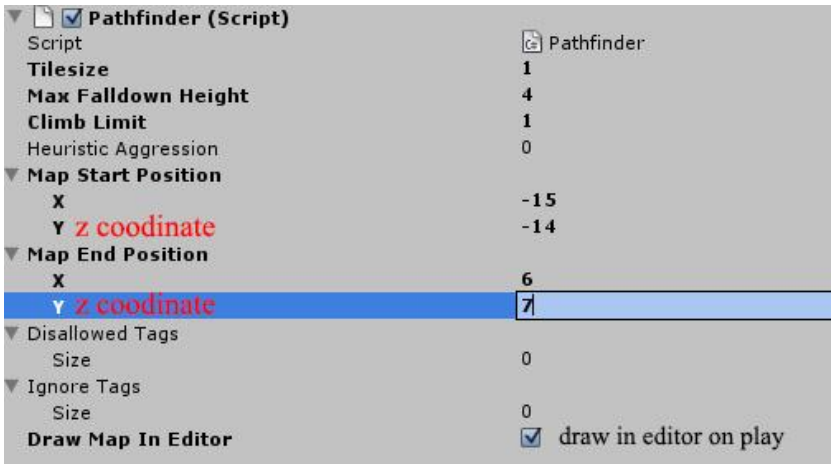
6. Map Start Position and **Map End Position** is the next step.

> ***NOTE!*** *The system only works with the Y axis being the up axis; the script needs to be changed if this is not desired. Because of this the Map start and end positions* ***(x, y) = (x, z).***

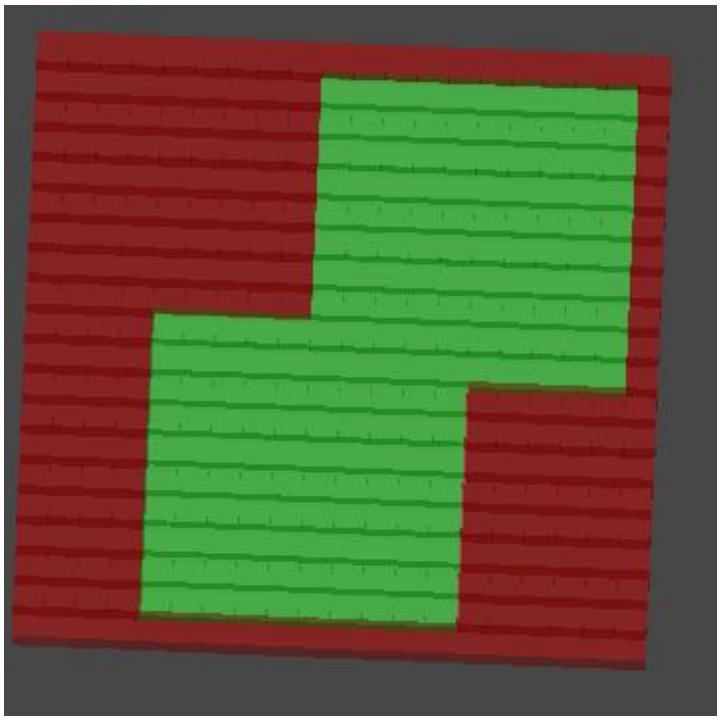
The positions should be setup according to your map:



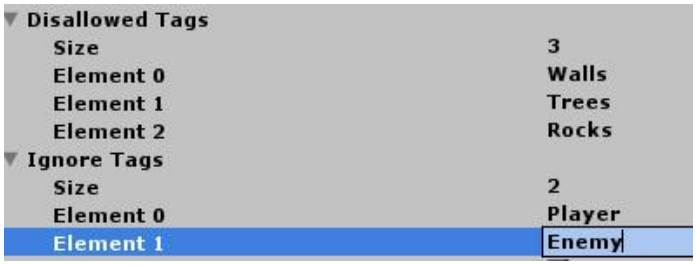
Make sure to remove any game object not being used (*like the spheres I used to find my positions with*). Try and choose **Draw Map In Editor** and watch the map upon play. It is always a good idea to add a margin to both the start and end position to make sure that the entire map is covered:

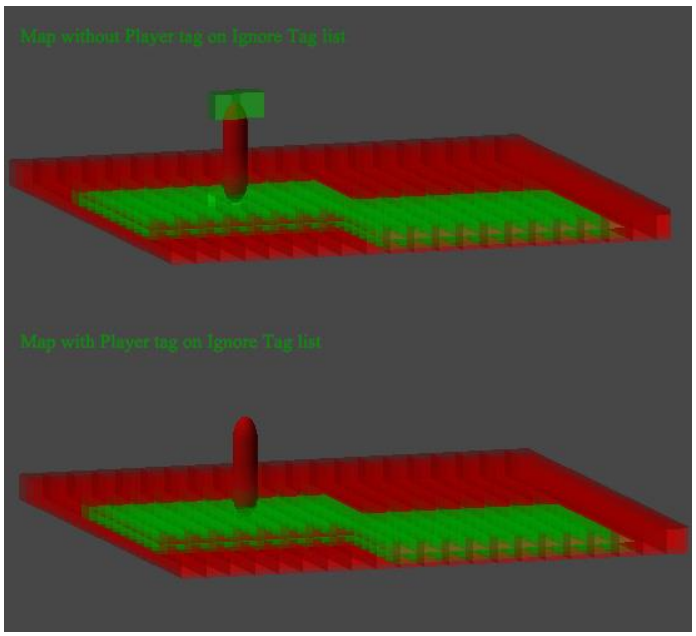


And see the map in play mode:

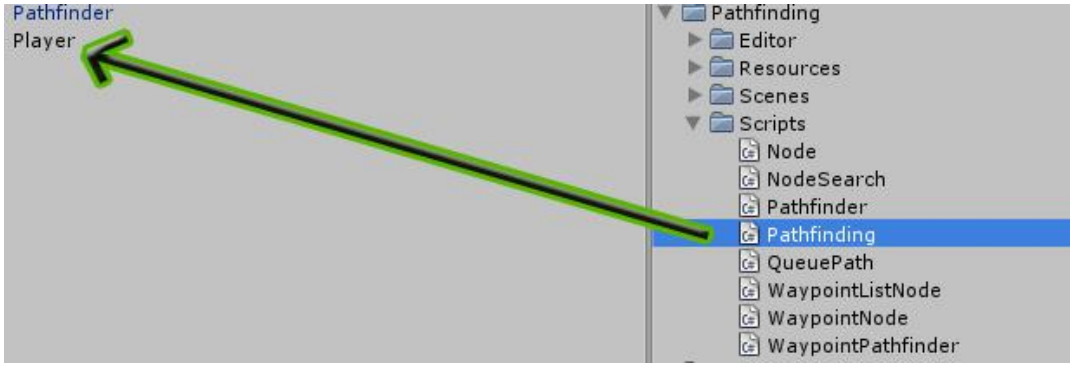


7. The last step is to set **Disallowed Tags** and **Ignore Tags**. GameObject tags are used to distinguish objects from each other. **Disallowed Tags** are used for static objects that should be non-walkable like walls, trees and so on. **Ignore Tags** are used for non-static objects that should not be part of the map such as player characters, enemies and more. To setup the tags, choose a size for each list in the inspector and simple fill out each column with the tag string. *(Remember to tag your objects accordingly!).* Walls higher then climb limit does not necessarily need to have a tag on **Disallowed Tag** list, but it is a good idea.

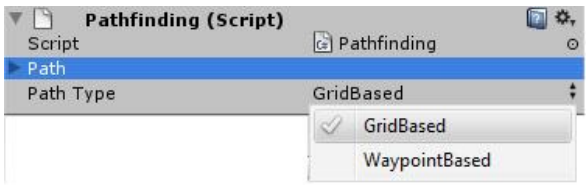




8. The setup of the **Pathfinder** is complete! However we need to add a **Pathfinding** script to all the objects we want to move accordingly to the map. The script is found at *Pathfinding/Scripts*.



After adding it, go to the **Inspector** and make sure that path type is set to **GridBased**:



9. In order to get the path a single method in the **Pathfinding** class needs to be called:

```
FindPath(Vector3 startPosition, Vector3 endPosition)
```

As an example:

gameObject.GetComponent<Pathfinding>().FindPath(transform.position, enemy.position).

However a great idea is simply to inherit the script in your player/AI scripts such that it can be called automatically from the script:

```
public class Enemy : Pathfinding
```

This is the optimal way as you have direct access to the path and path call! An example class is:

```
public class Enemy : Pathfinding
{
    Vector3 endPosition = new Vector3(90, 3, 87);

   void Update ()
    {
        //If i hit the P key i will get a path from my position to my end position
        if (Input.GetKeyDown(KeyCode.P))
        {
            FindPath(transform.position, endPosition);
        }
        //If path count is bigger than zero then call a move method
        if (Path.Count > 0)
        {
            Move();
        }
    }
}
```

The path list<> is called **Path**. A simple example of how to use it to move a character can be found in the **Pathfinding** class:

```
    //A test move function, can easily be replaced – call it to test the map!
    public void Move()
    {
        if (Path.Count > 0)
        {
            transform.position = Vector3.MoveTowards(transform.position, Path[0],
            Time.deltaTime * 30F);
            if (Vector3.Distance(transform.position, Path[0]) < 0.1F)
            {
                Path.RemoveAt(0);
            }
        }
    }
```

In most cases something more advanced will be needed, so simply write your own function using the **Path** list<>.

GridBased Notes:

Now that the grid based pathfinding system is setup there are a few things to consider. For example it can be used for procedural generated maps. However the map is created in the **Start()** method and therefore objects for the maps should be instantiated in the **Awake()** method. Furthermore the Start and End position should be known and set in the **Awake()** method as well if the size of the map changes:

```
void Awake()
{
    Pathfinder.Instance.MapStartPosition = new Vector2(5, 10);
    Pathfinder.Instance.MapEndPosition = new Vector2(250, 370);
}
```

The grid based solution can be used for many types of games. However it does not work on maps with different *"levels"* like houses with multiple floors and such. If this is needed use the **WaypointBased** solution described next.

ADDING DYNAMIC MAP ITEMS TO THE GRID BASED SOLUTION

With the 1.01 Update, it is now possible to add a script to a dynamic update which will change the grid based map at runtime.

All you have to do is adding the script **DynamicTDGridObject**, found at *pathfinding/scripts/DynamicTDGridObject.cs*. Note that it can be used for all type of games, not only tower defence games, but fits really well for tower defence games.

Important: The calculation requirements increase for larger objects as the check is based on bounding boxes. Therefore do not update larger items too often. If you are going to update more complex objects divide it into smaller chunks, like a fallen tree could be divided into checking for the leaf-crown and the log.

In order to decrease how often it updates, we can use a coroutine! The class already contains the method. All you have to do is set **SetTimer** to *true*, and then define **Timer** (*cannot be done runtime, set it before*).

In our example it will update it five times per second. If we set **SetTimer** to *false* it will update every frame if the object changed position or rotation.

Update:

You can now also use the *DynamicUpdateMap.cs* script. It will redraw part of the map with raycasts. This makes it possible to add dynamic objects such as bridges and such which will make new parts of the map available. All you have do is set how often it should update in the inspector. The larger the object the slower it will update. It should however be pretty fast, and might be better to use than the DynamicTDGridObject.cs.

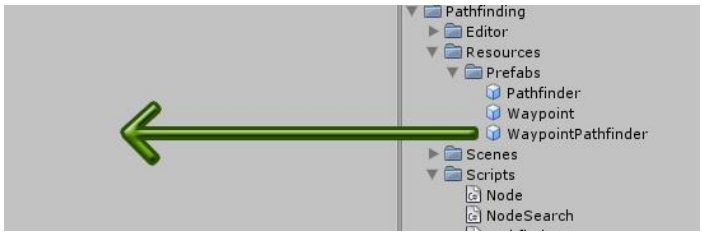
THE WAYPOINT BASED SOLUTION

The waypoint based solution works rather different then the grid based solution. First of all it is built in the editor by the level designer. It works with different "levels" like houses with several floors etc. A map is created by placing waypoints and connecting them manually with different mouse/keyboard controls. The project comes with five different controls:

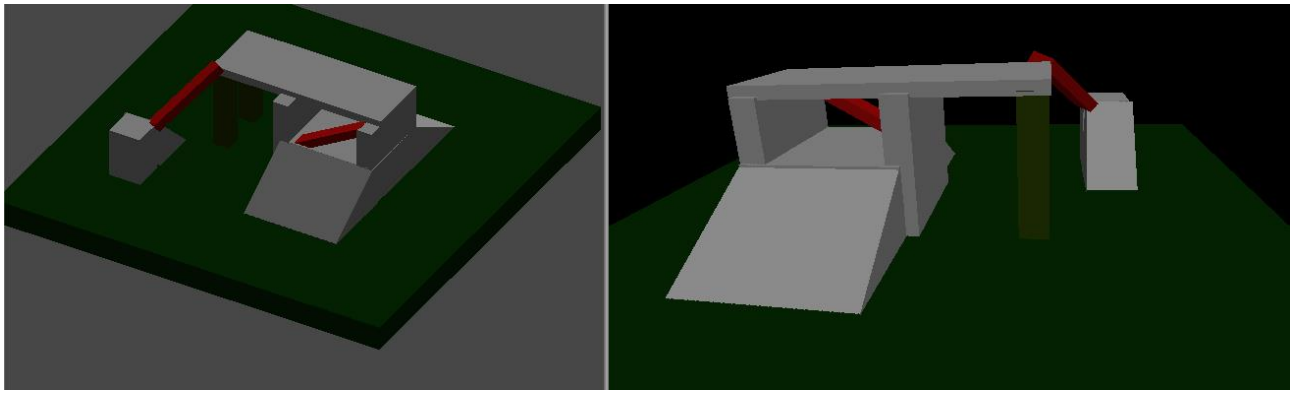
1. **Right mouse click:** By clicking on an empty spot a **NEW** waypoint is created. If a right mouse click is performed on an existing waypoint it will become the **ACTIVE** waypoint. An active waypoint is the one we connect to other waypoints.
2. **Right mouse click + CTRL:** By clicking on a waypoint that is not the active waypoint, the active waypoint AND the clicked waypoint will **connect** with each other in **BOTH** directions.
3. **Right mouse click + CTRL + ALT:** Using this combo on a waypoint will **DELETE** it.
4. **Left mouse click + ALT:** This combo will **disconnect** the path between the active and the clicked waypoint, but ONLY in the active waypoints direction. This is useful if for example it should be possible to jump down from a tall rock but not climb it.
5. **Left mouse click + CTRL + ALT:** This combo will **disconnect** the path between the active node and the clicked waypoint in **BOTH** directions.

All of this will make more sense when we go through the setup of a simple level. ->

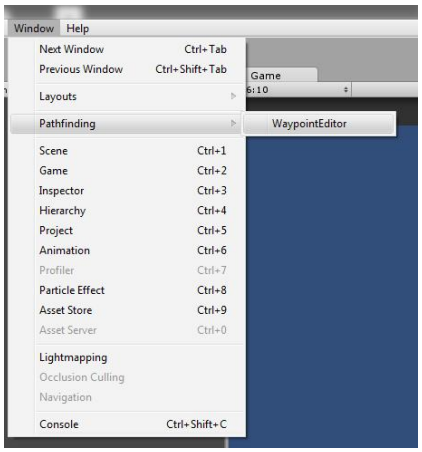
1. Drag the **WaypointPathfinder** Prefab into the scene, it is found at *Pathfinding/Resources/Prefabs.*



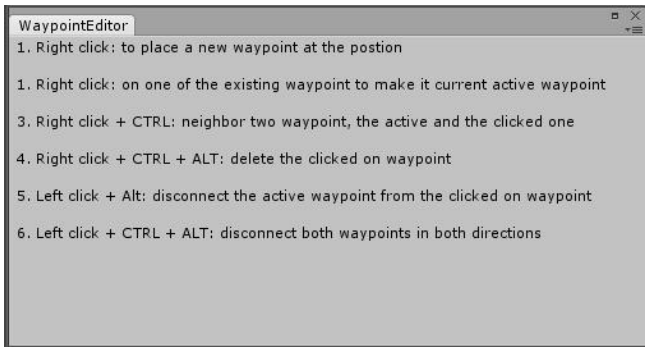
2. Now build some type of level (*example is shown here*):



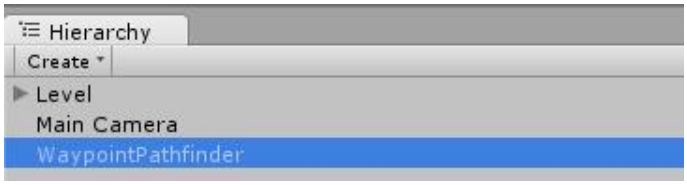
3. Find **Pathfinding** in the menu under **Window** and pick WaypointEditor, *it might take some time to show after importing*:

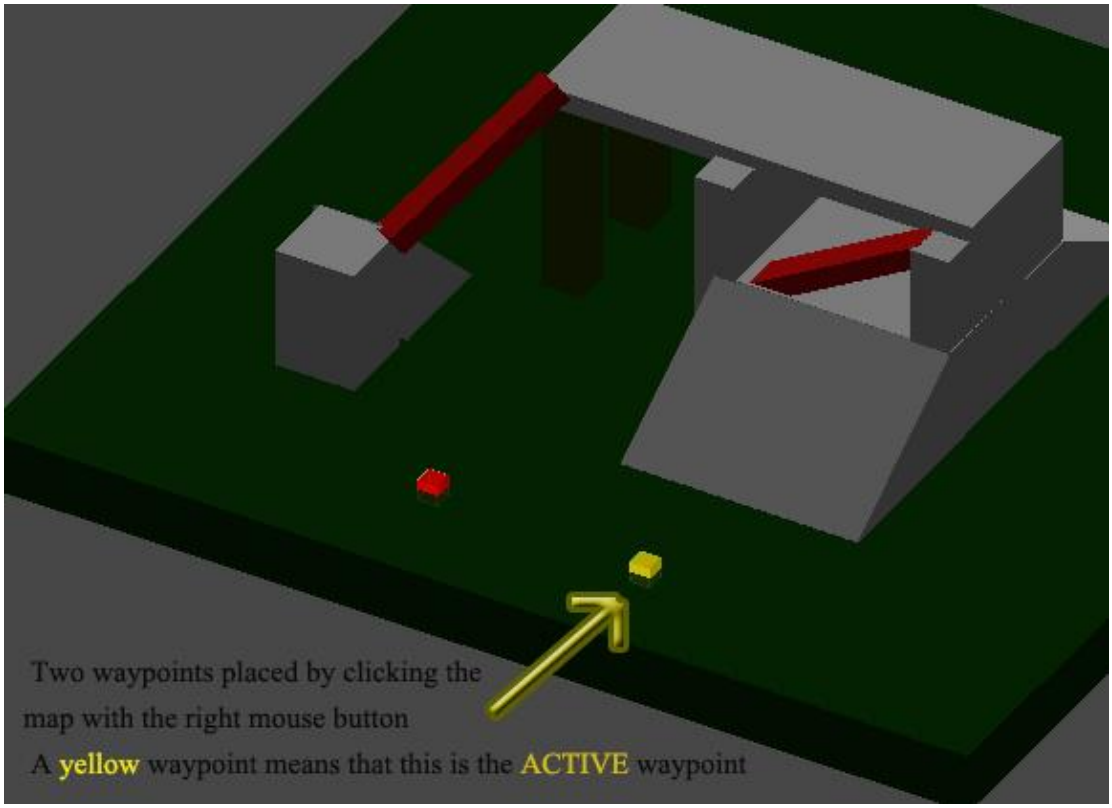


And you will get a window like this:

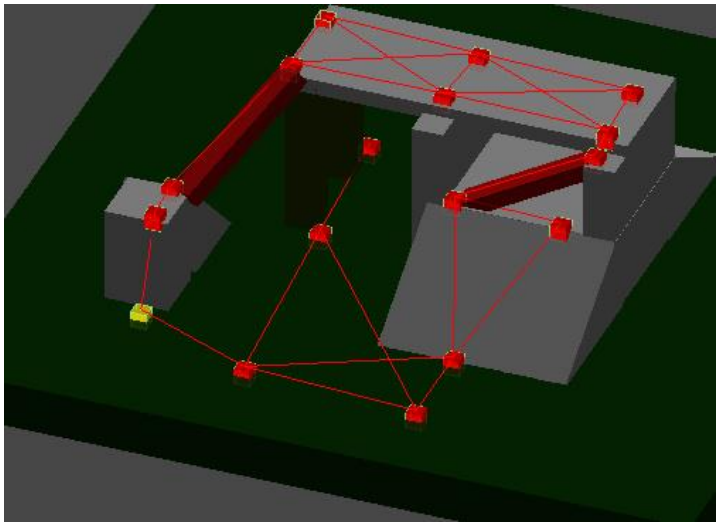


4. Now click on the **WaypointPathfinder** Prefab in the scene hierarchy and start placing waypoints with the right mouse button, the map will **ONLY** be shown when the **WaypointPathfinder** Prefab is selected:

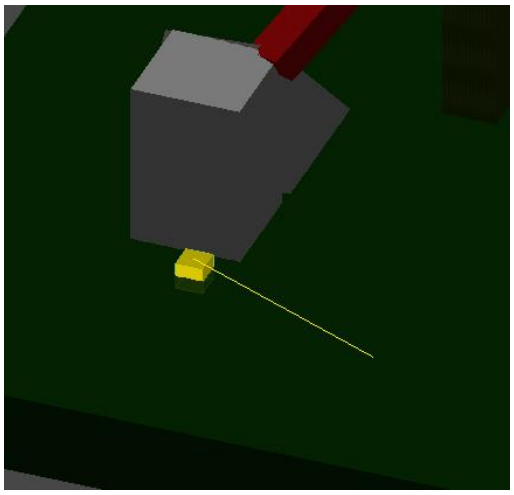




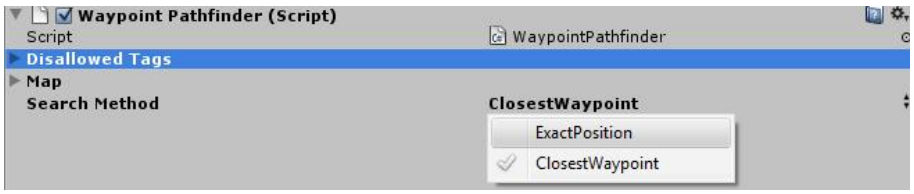
Now connect the active waypoint with another waypoint with **Right mouse click + CTRL**. Remember you can change the active waypoint by right clicking another waypoint. You well then get a map like this:



Sometimes you only want to go one way but not the other; this is possible by **Left mouse click + ALT**. As seen it the picture, compared to the last picture, it does not connect upwards anymore:

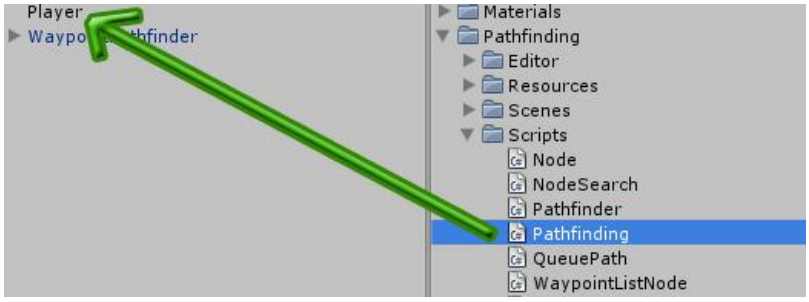


5. Now you need to choose one of two search method types to use, you set it under the **WaypointPathfinder** prefab. The **ExactPosition** method will go to the exact position that you choose as end position. Or you can choose to only go to the nearest waypoint; this is done by choosing the **ClosestWaypoint.**

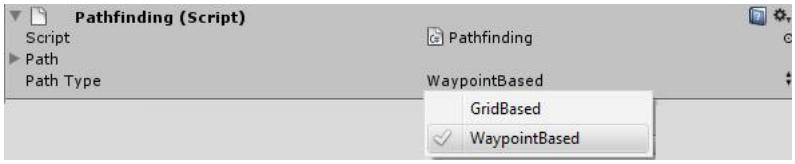


If you use the **ExactPosition** method there might be spots you don't want to go to. This could be the case in games where you can click and find the end position with the mouse *(like house walls)*. In order to fix this you can set **DisallowedTags** just as we did in the grid based solution. *This is still a bit buggy and will be fixed as soon as possible*.

6. The setup of the **WaypointPathfinder** is complete! However we need to add a **Pathfinding** script to all the objects we want to move accordingly to the map. The script is found at *Pathfinding/Scripts*.



After adding it, go to the **Inspector** and make sure that path type is set to **WaypointBased**:



7. In order to get the path a single method in the **Pathfinding** class needs to be called:

```
FindPath(Vector3 startPosition, Vector3 endPosition)
```

As an example:

gameObject.GetComponent<Pathfinding>().FindPath(transform.position, enemy.position).

However a great idea is simply to inherit the script in your player/AI scripts such that it can be called automatically from the script:

```
public class Enemy : Pathfinding
```

This is the optimal way as you have direct access to the path and path call! An example class is:

```
public class Enemy : Pathfinding
{
    Vector3 endPosition = new Vector3(90, 3, 87);

   void Update ()
    {
        //If i hit the P key i will get a path from my position to my end position
```

```
        if (Input.GetKeyDown(KeyCode.P))
        {
            FindPath(transform.position, endPosition);
        }
        //If path count is bigger than zero then call a move method
        if (Path.Count > 0)
        {
            Move();
        }
    }
}
```

The path list<> is called **Path**. A simple example of how to use it to move a character can be found in the **Pathfinding** class:

```
    //A test move function, can easily be replaced – call it to test the map!
    public void Move()
    {
        if (Path.Count > 0)
        {
            transform.position = Vector3.MoveTowards(transform.position, Path[0],
            Time.deltaTime * 30F);
            if (Vector3.Distance(transform.position, Path[0]) < 0.1F)
            {
                Path.RemoveAt(0);
            }
        }
    }
```

In most cases something more advanced will be needed, so simply write your own function using the **Path** list<>.

WaypointBased Notes:

The waypoint based approach shines for different reasons. It can be extremely fast because it can be limited to only use the needed important navigation spots. Secondly it can work on more “levels”, like houses with multiple floors and such. It gives the level designer complete controller of the design without coding at all. However it cannot be used for procedural generated content.

JAVASCRIPT (UNITYSCRIPT) SUPPORT:

In order to use the system with JavaScript we need to add two scripts to each AI, and tell it to use JavaScript.

The first script we need to add is **Pathfinding.cs** found at *Pathfinding/Scripts/Pathfinding.cs.* Next step is to either add **JSPath.js** found at *Pathfinding/Scripts/JSPath.js.* **OR** extend the JSPath class in another JavaScript like:

```
class JSGridPlayer extends JSPath
{
	// Class in here
}
```

An example of this can be found in the **JSGridPlayer.js** found at *Pathfinding/Scenes/Scripts/JSGridPlayer.js.* To test how it work open the scene **JSQuickScene** in the scene folder.

Last thing we need to is to tell our Pathfinding.cs class that we are using JavaScript:

2D SUPPORT IN THE XY PLANE:

In order to use it for 2D games in the XY plane, first we need to add the prefab Pathfinder2D:

Remember that in order to draw a map, it will need colliders on objects.

1. Drag the **Pathfinder2D** Prefab into the scene, it is found at *Pathfinding/Resources/Prefabs.*

Then we need to set zStart and zEnd in the inspector! The camera direction should be towards a positive Z value. For example the camera should be at -10 z, look at [-10, infinity] on the Z axis.

2. Now we need to add the script Pathfinding2D to our agents! Found at *Pathfinding/Scripts*. We can also simply use a script which inherits this script!

If we are going to use JS then we still need to at this script and the **JSPath.js** found at *Pathfinding/Scripts/JSPath.js.* Or and inherited script of this one. (*See JavaScript support section*).
There is a simple 2D test scene in the scene folder, and a few simple AI scripts all with the 2D extension in its name.

CONTACT BFGAMES:

CONTACT MAIL: bfgamess **AT** gmail **DOT** com

VIDEO CHANNEL FOR TUTORIALS: http://www.youtube.com/channel/UCYze0NNv-T5m_SAeYfTk_Sw

UNITY FORUM THREAD: http://forum.unity3d.com/threads/164003-BFGames-Simply-A*-Pathfinding!

THANKS FOR YOUR INTEREST IN SIMPLY A*!