Navigation And Pathfinding System User's Guide

Content

Introduction

What is NAPS?

Navigation And Pathfinding System allow you to quickly make navigation for any type of AI at your level. NAPS combine the ease of use of the so-called waypoints and the ability to describe large areas which makes navigation mesh. With this you get excellent performance which is important when using the navigation at the level where are a large number of AI exemplars. Using functions of path-finding library you can build any path-following algorithm from simple point-to-point following to a complex movement system based on the visibility of certain areas, taking into account optimal path, cutting corners, resumption of route in case when loss a visibility of point, etc. Another important feature of the NAPS is combining of two algorithms of calculation of paths, one of them precalculate most part of navigation data. This mean that you don't need to perform complex calculation in paving the route from one point to another as you have to do in case of using A * algorithm. The basic calculation is carried out only once in case of changing the structure of the paths grid, this mean that you can use a huge number of units which are at the same time paving the way for themselves with minimal loss of productivity. At same time all navigation data stored in files that you can dynamically load / unload for use in different parts of the map or different paths for desired AI type. This is useful when working with very large territories and loading of all navigation data at the same time is too expensive. For more information examine attached examples of the movement AI as well as the functions of the path-finding library (Pathfinding.cs).

Navigation in NAPS

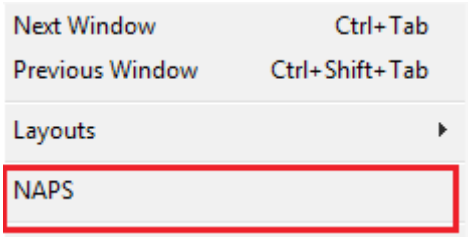
Navigating in NAPS is due to the grid of paths consisting of nodes. Between any two nodes of the grid can pave the shortest route. In order to separate the movement of different types of AI common network of paths can be split up into paths subnets then a certain type of AI can specify the valid subnet which it can use for movement, not specified subnet will not be used by this type of AI. So how made this subnets? We have already said that the network consists of nodes. Each node has a type, type is just a set of symbols, that is a string representation of any name you come up with can serve as a type of node. The union of nodes that are of same type represent a subnet. Each node has a unique serial number inside his subnet. Let's say in your game you have two kind of creatures: peoples and dragons. Let the people have the opportunity to walk only by land routes (stairways, rooms, roads, etc.). Dragons can also use some of the human routes when they are not in the air, but the dragons should be also able to fly on his special "air" paths as opposed to people. Thus it is advisable to create two types of subnets of paths, one - only for the people,

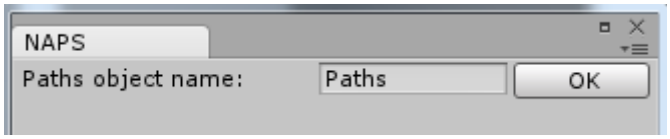
second - only for the dragons. We call subnet used by people - «human», a subnet used by dragons - «dragon». Thus by placing on the ground nodes of the type «human» we will create a subnet of land routes, subnet «dragon» we will create by placing in the air nodes of the type «dragon». And so, we have two separate subnets that can be used separately. But then we need to dragons could be use not only subnet «dragon», but also a subnet «human». To do this, create transitions between these subnets by connecting some of their nodes, more information about this in the section "Atypical connections." Advantages of subnets is in the fact that each subnet routes calculated beforehand and not in real time. This mean that the AI that can use multiple subnets as the dragon, which was mentioned above, does not require a complex calculation of the route as it uses the data already calculated. When writing the movement AI you can determine what type of AI can use one or the other type of subnet (see the examples of AI).

Work with NAPS editor

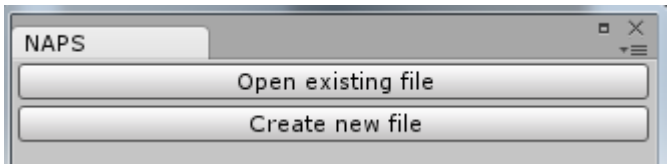
Getting started

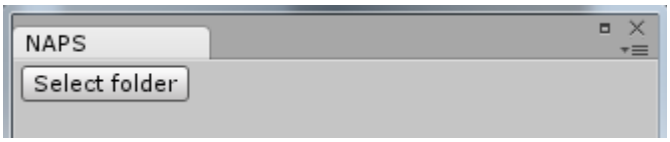
In order to open NAPS window, execute Window\NAPS in editor's main menu and you will see NAPS window.



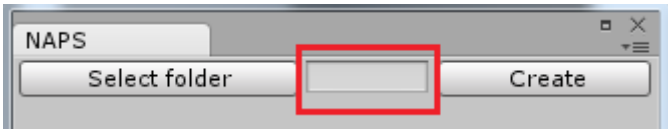


In the field «Path object name» by default written «Paths» - this is name of the object that will store the objects of all nodes of the grid of paths in the scene. You can change it if the name is already being used for some object in the scene because the name must be unique. After pressing the button «OK» file dialog will be opened.

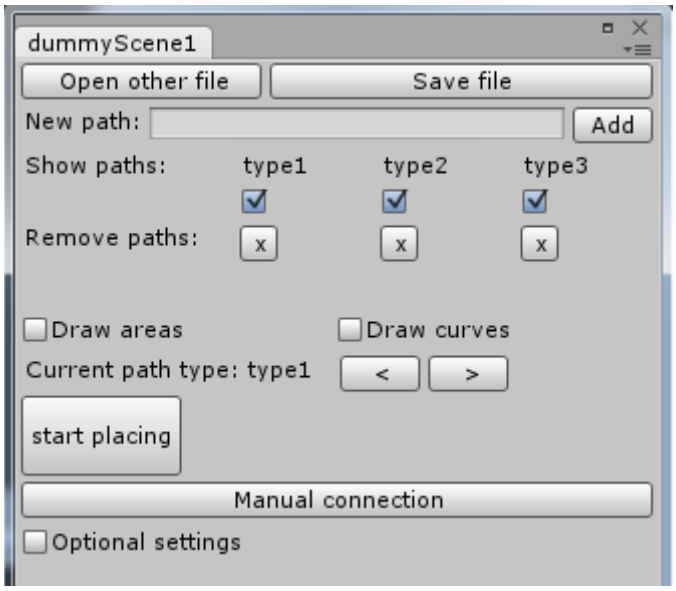




If you already have a navigation file you can open it by clicking on «Open existing file». Otherwise click «Create new file» in order to create a new navigation file. In this case you will see a button «Select folder» by pressing it a dialog of folder selection will be opened to save file. Once the folder for saving file was selected, you will see a dialog box for entering the file name.

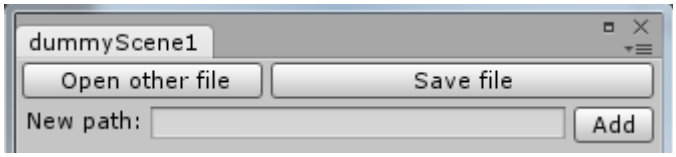


Enter the file name in the box and click «Create». When you create a new file NAPS actually creates three files, two of which have the same name. One of them contains only navigation data and has the extension *. nvf, this file is primary when working with navigation. The second file contains the color information of subnets visualization in scene view. This file has the extension *. est and is a subsidiary because of navigation file can work without this file, but the color information of subnet visualization in scene view will be reset to default. The third file - this is the custom settings file, it retains all the advanced settings of NAPS editor except visualization colors of paths subnets. This file called «Settings» and has the extension *. wst, by default located in the root directory of NAPS. After opening an existing file or create a new, you will see the main menu of NAPS.



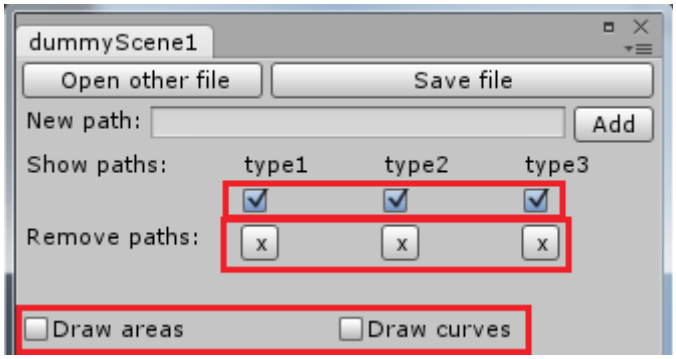
Path types. Creation, visualization, deleting.

In order to create a new subnet of paths enter the name (type) in the field «New path» in the upper left corner of the main window of NAPS and click button «Add». Will be created a new empty subnet of paths which will be able to add nodes.



As the name of the subnet type can be any combination of numbers and symbols, but for convenience I recommend giving them informative names such as «human» which shows that this type of path mainly used by peoples or similar creatures.

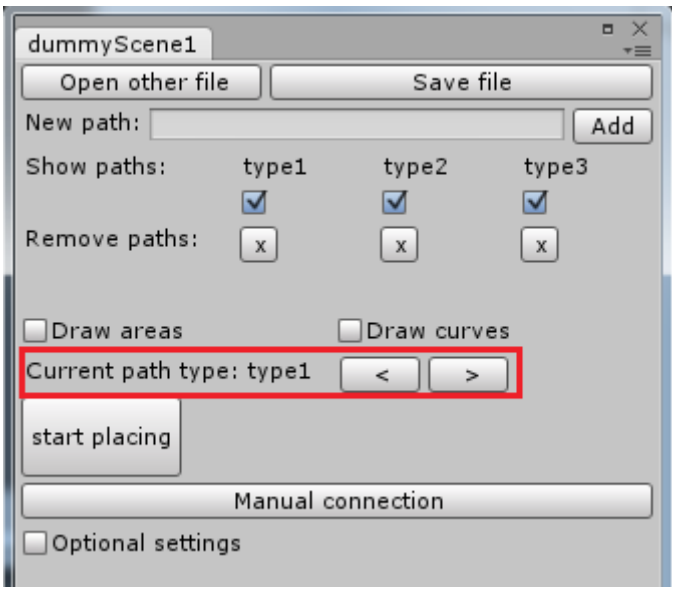
After that as one or more subnets will be created the options of visualization of subnets will be able. Here you can turn off the display of all or only some subnets by removing / setting the check-box below the name of the subnet.

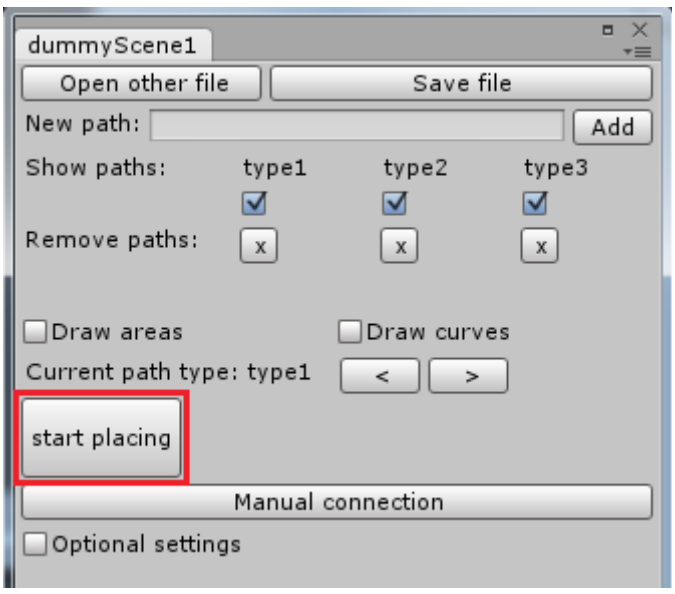


If one of subnets of paths is not needed anymore then you can remove it by pressing «Remove paths» button under this subnet. You can also enable / disable the visualization of areas of displacement and the curves by setting / clearing the check-box «Draw areas» and «Draw curves» respectively. Visualization of a large number of connections of nodes can greatly reduce the number of frames per second in the editor so for a comfortable work I recommend to disable of drawing of subnets not used at the moment.

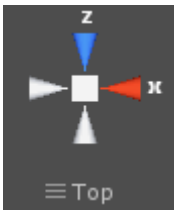
Creation of paths nodes

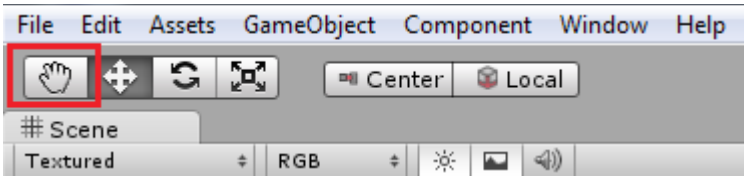
Once you have added one or more subnets you can create nodes inside one of them. To do this select the type of the subnet in which you want to create a node, you can do this by pressing the arrow keys in the field «Current path type». Current type of subnet of paths will then be displayed to the left of the button (or the right of the field).



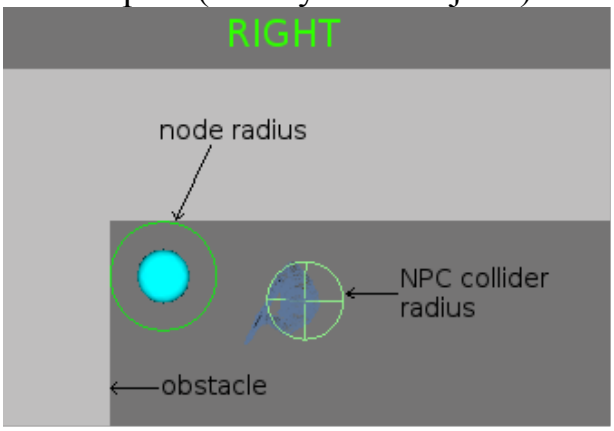


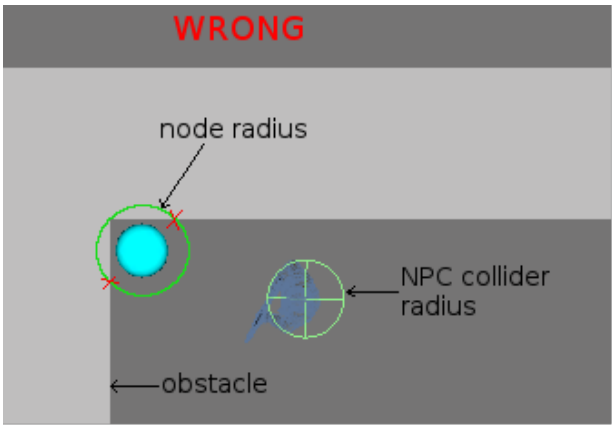
Once the desired subnet has been selected press the «Start placing» button. For convenience expand the scene view to full screen (press "space" by default), in order to locate the point use direction of view of scene camera. If you plan to pave the paths on an open area, not indoors, then you can turn on the top view (in the top right corner of scene view click on the arrow pointing to the top of the box) and enable an orthogonal view (click on the box itself).



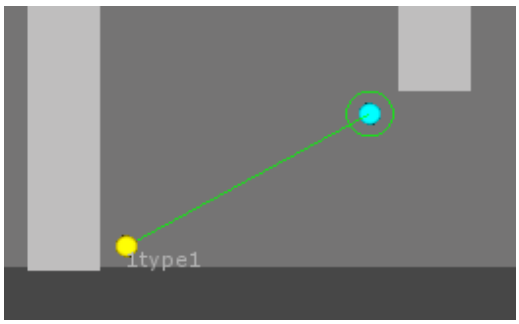


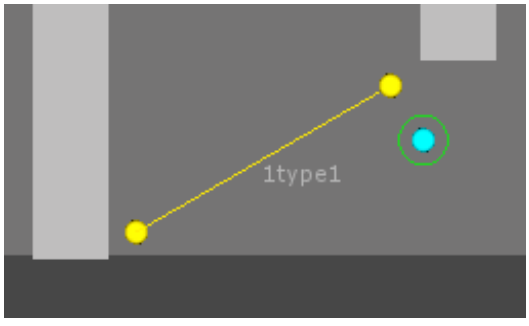
You can then move the camera with the tool «Hand». When the camera view will be directed to the object with designated collider you will see in the center of screen a cycle gizmo on the surface of the object. The radius of the circle indicates the radius of the point, this radius must be equal to or slightly greater than the radius of collider of character which will be able to move through this point. You can adjust the radius in the optional settings of NAPS window. This circle is necessary only for the correct location of the points so be careful that it never has been blocked by other objects through which the character will not be able to pass (usually static objects).



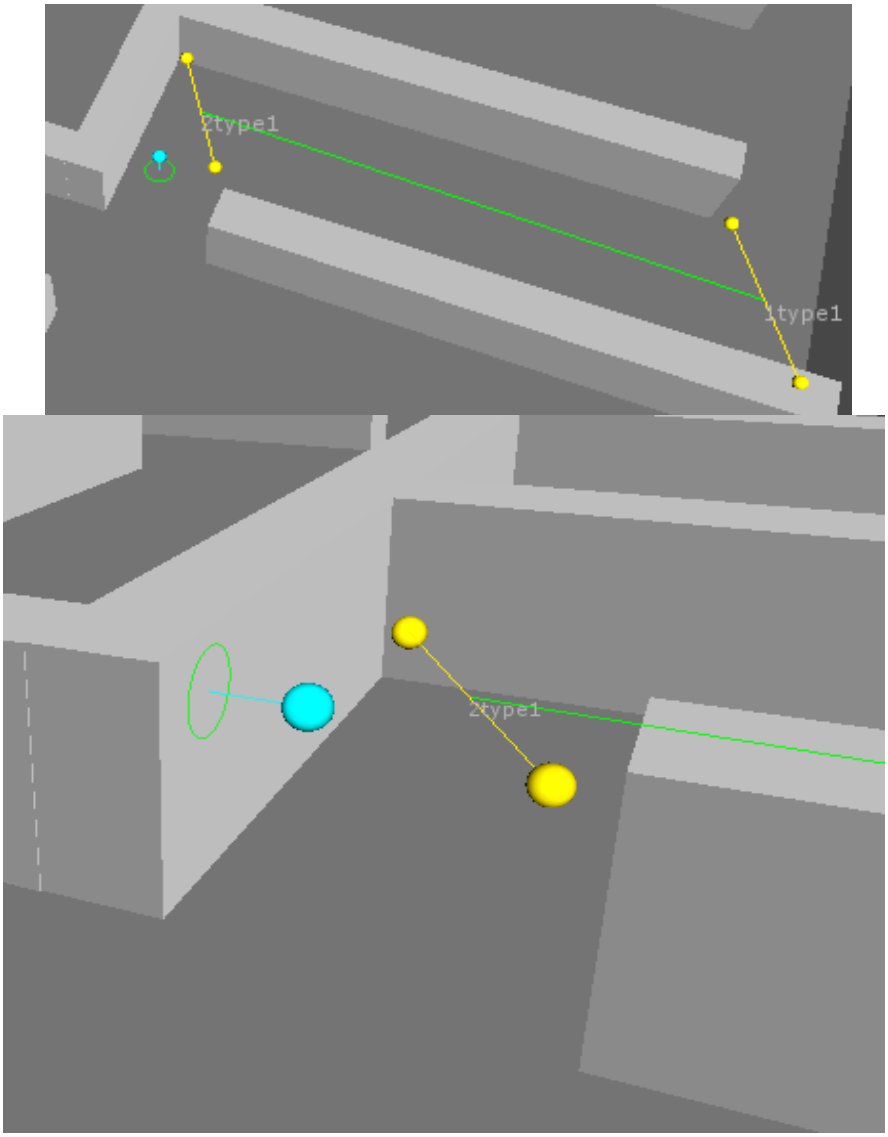


You can place a single points as well as bipoints. In order to accommodate a single point, in the right place press key "1" then key "2." You will see the created single point. To place bipoint press key "1" at the first point then move the camera to the location of the second point and press again key "1", and you will see created bipoint, between the points of which, passes the line the color of which you can set in optional parameters (for more details about bipoints see section "What is bipoints and how to use them?").





Single points it makes sense to put in places where you do not need a variance path, for example - a narrow alley. It is enough to put one single point at entrance and one more at the exit of alley because within a narrow aisle NPC will have still move by a straight line. If on the contrary, the point must be placed on a wide open street so that the NPC was able to use the maximum space to move - you should use bipoints. To locate nodes inside an enclosed space it's best to use scene camera mode «Fly». To do this turn the scene camera to perspective view (click on the box in the upper right corner of the scene view). After this hold down the RMB and by using W, A, S, D keys you can "fly" a camera as in FPS game, but instead of aim you will see the position of current point.

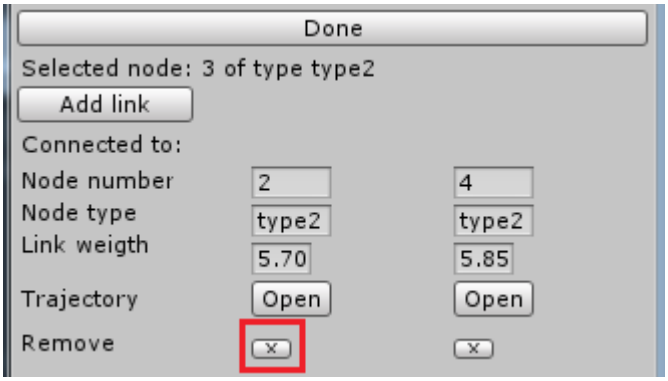


Any newly created node will be automatically connected with the other visible nodes of the same type. You can adjust the position of nodes using the «Move»

tool. To do this simply click on the one of points of desired node which are displayed in scene view and move it like an any other object.

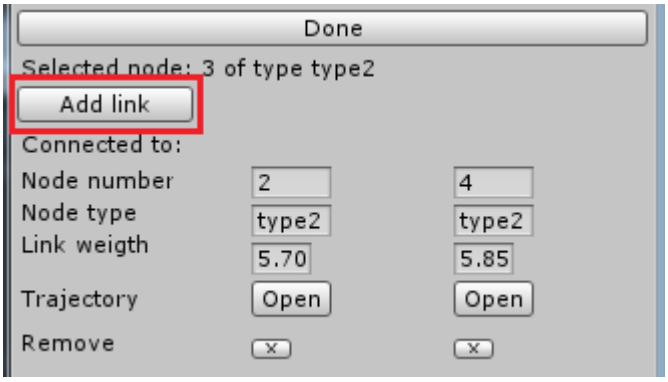
Work with connections of nodes

When you create new nodes in some subnet of paths, they are automatically connects with all visible nodes of that subnet. The connection between nodes can be one-sided or two-sided. Two-sided connection between two nodes - is a connection in which is a pass from one node to the other node and vice versa. One-sided connection between two nodes - is a connection in which you can pass from one of the nodes to another, but not vice versa. In some cases it is useful if you want to, for example, create a path or part of the path with only one-way traffic. For each node, created automatically, sets two-sided connections with all visible nodes of its subnet. In order to remove unnecessary links or add new ones select by mouse click desired node in scene view and press button «Manual connection» in main window of NAPS. You will see a dialog to work with the connections of nodes. If you need to remove certain links of this node with other nodes, then have a look at the list of those present connections with other nodes under the heading «Connected to». Next find the deleted node in the list and click «Remove» under his data.

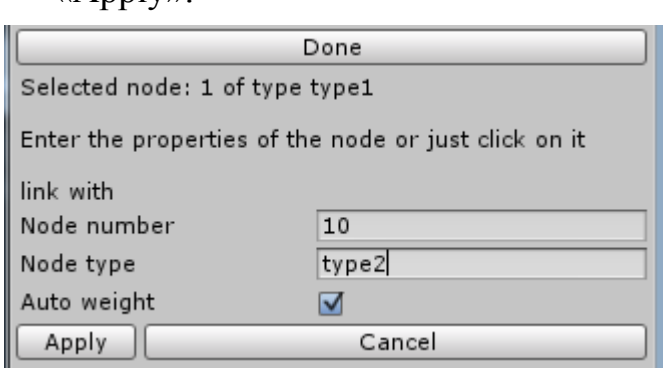


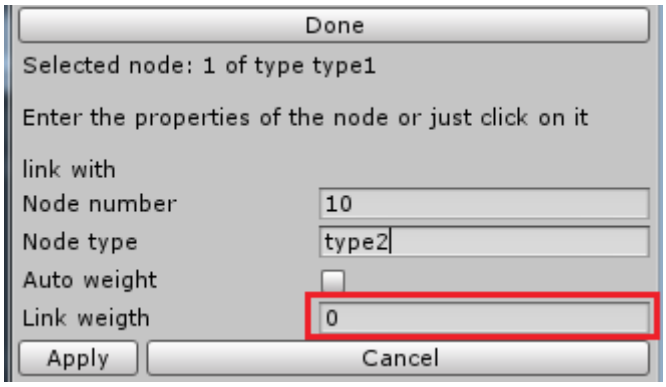
But remember that with this you only remove link of this node with the other, but not vise versa. That is to pass directly from the current node to the second node is no longer possible, but can pass from the second node to the current node (one-sided connection is still). In order to completely remove any links between these nodes you must also remove current node from the list of active links of the second node.

To add a new link of this node with another node click «Add link».



In the menu that appears enter the number and type of node you want to connect to or just select this node by LMB click in scene view. After this click the button «Apply».

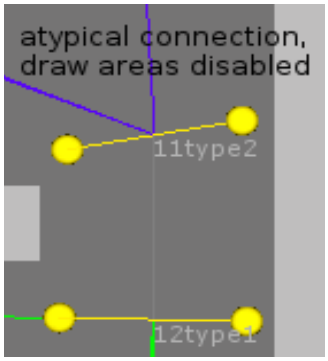


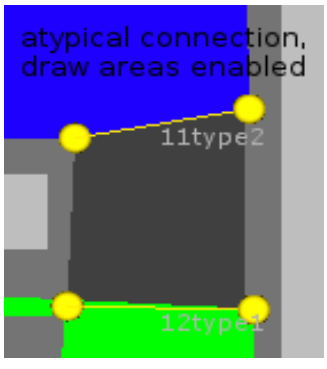


If the check-box «Auto weight» is set then the weight of the connection is established automatically (the distance between nodes will be taken as value), otherwise you can enter any desired value. To cancel adding a new connection press «Cancel».

Atypical connections

In order to NPC can use several types of subnets of paths it is necessary to create atypical connections between nodes of this subnets. These connections serve as a "bridge" between certain types of subnets of paths, they shows from what nodes of one subnet can pass to some nodes of any other subnet. To understand this you can, once again, to give an example of people and dragons. Let's have a subnet called «human» and people can use all the nodes if that subnet. There is also a subnet «dragon» - air paths used only by dragons. But dragons also need to be able to go down to the ground and move by some land routes. Thus it turns out that the dragons, unlike the people, should be able to use two types of subnets. And here we are helped by the atypical connections. Atypical connections - this is a common one-sided or two-sided connection between nodes of different subnets.



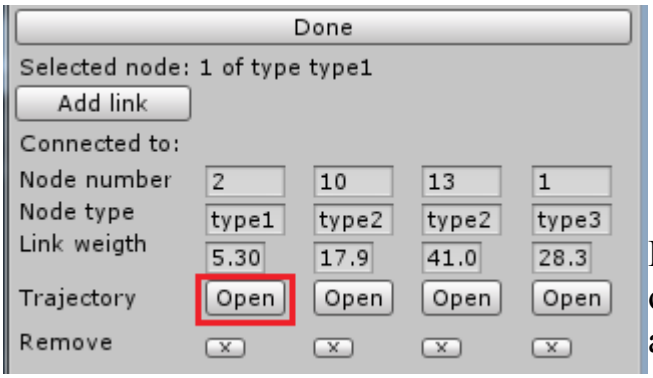


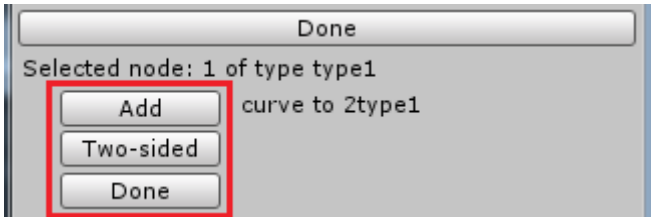
So with their help we can combine several different subnets into one. In this case the dragon will be aware about nodes of its subnet («dragon») from which can pass to the ground subnet («human»).

Work with curves

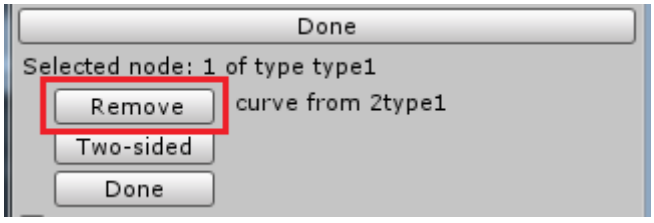
In NAPS you can add Bezier curves to the existing connections between the nodes. It can be useful, for example, if you want to create cinematic scenes in your game. For this purpose between the nodes sets a curve at which you can move any object, in this case - the camera. Then you can record a sequence of nodes connected by curves as a route for the camera creating a cinematic flight.

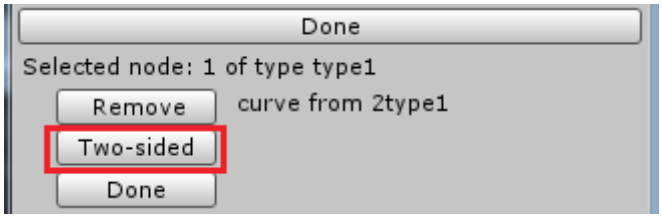
In order to add a curve between a pair of nodes it is necessary to between these nodes had a connection. If the connection between the nodes exists it is possible to add a curve. To do this select the node which you want to connect by curve to another node, then go to the menu «Manual connection» to work with the connections. In the list of connections find the next node to which you want to connect by curve selected node. When connection to the second node does not exist then you can add it with «Add link» menu as conventional connections. When the connection is there you can see the connection parameters in the field «Trajectory» and click «Open» in front of it.

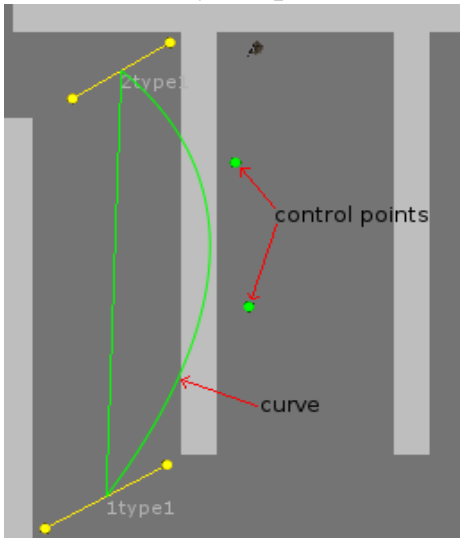




By pressing the button «Open» you will open a dialogue to work with curves. To add a new curve to the current connection click «Add». In order to display the curve in the scene set check-box «Draw curves» in the main menu of NAPS. When new curve added at the central points of connected nodes appear small spheres similar to the spheres that represent points of nodes – this is the control points of the curve, by moving them you can ask any shape of the curve.



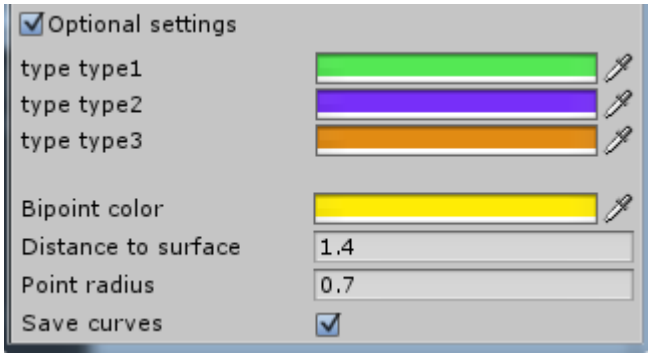




If you need to make this curve also available from the second node (two-sided connection), you can do this by clicking the button «Two-sided» or you can manually add the same curve between the second node and current node. If the curve is added to the current connection, the dialog of work with curves instead of the «Add» button will displaying «Remove» button by pressing at which curve will be removed from this connection. To exit from the dialog of work with curves click «Done».

Optional settings

In the menu of optional settings you can change parameters of rendering of the grid of paths in the scene, as well as some other options. To open the menu of optional settings set check-box «Optional settings» in the main menu of NAPS.



At the top of the menu there are fields with the name «type» in front of which are the names of subnets of paths as well as customizable color fields. These parameters are responsible to colors of displaying of corresponding subnets in scene. You can work with them as with conventional color fields. These settings are stored in the file with *.est extension. Under the subnets colors settings is located field «Bipoint color», this parameter is responsible for the color of the displaying of points of the nodes of the grid of paths. Parameter «Distance to surface» indicates the distance at which the node will be located by default from the surface at which it was placed. This parameter need only for the convenience of placing nodes on a fixed distance from the surface, if necessary you can change the position of the node in the editor using the tool «Move». Parameter «Point radius» responsible for rendering the diameter of the currently placed node

directly when placing, i.e. after pressing «Start placing». In order to correctly place the node you need to set the «Point radius» equal to the radius of the collider of character that can move through this node + a small increment for the insurance. For example the radius of the character's collider equal to 0.5 m, then the parameter «Point radius» should be set to approximately 0.6m. Then it is necessary to place node in the way in that the cycle displaying in the center of screen have been not overlap impassable obstacles.

Check-box «Save curves» is responsible for the preservation of these curves in the current navigation file you are editing. If it set then all data of the curves will be saved along with the rest of the navigation data to a file. If you do not need to use the curves in the current navigation file then clear the check box as this will reduce the file size, if its size is so large that it matters. Otherwise if you do not use the curves in the current navigation file but the flag «Save curves» is set then inside navigation file will be reserved storage space for curves data which slightly increases the size of the file. All parameters except the colors of displaying of subnets are stored in a file Settings.wst in the root directory of NAPS.

Work with navigation files and dynamic data loading

The navigation data of NAPS stored in a file with the extension *. nvf. When data is saved the NAPS editor also creates a file with the extension *. est which store only settings of displaying of subnets of paths. In case when *.est file was broken or lost you can continue work without it, because when working with navigation is only important a file with extension *.nvf. In NAPS you can dynamically load the navigation data, for this purpose path-finding library (Pathfinding.cs) has function of loading data - LoadNavigationData.

static function LoadNavigationData(string filePath, string fileName) : Void

In order to download data simply call this function with the path to the file as filePaths and the file name as fileName, you don't need file extension for this.

Example:
Pathfinding.LoadNavigationData(“D:\NAPS\Paths”, “newMap”);

Another useful feature when working with navigation data is function DataLoaded of library Pathfinding.cs. This function returns a Boolean, true - if the navigation data is loaded; false - otherwise. You can create a lot of navigation files for your game level if it is too huge and then upload the files

with required portions of the data. Below is a code of script PathfindingManager.cs which is responsible for loading the navigation data .

Code:

```
using UnityEngine;
using System.Collections;

public class PathfindingManager : MonoBehaviour {
     //contains file name
     public  string fileName = "dummyScene1";
//contains file path
     public  string filePath = "D:/Unity
Projects/NAPS/Assets/NAPS/Paths";
/*determines whether can autoload navigation data, when loading
scene
*/
     public bool autoload = false;
//user interface check-box
     bool newFilePathDialog = true;

     void Awake(){
//if autoloading is prohibited,
          if(!autoload)
// then stop execution and return
               return;
//else, if navigation data is not loaded yet,
          if(!Pathfinding.DataLoaded()){
//then load navigation data
          Pathfinding.LoadNavigationData(filePath,fileName);
          }
     }

     void Update(){
//if autoloading is prohibited,
          if(!autoload)
// then stop execution and return
               return;
//else, if navigation data is not loaded yet,
          if(!Pathfinding.DataLoaded()){
//then load navigation data
          Pathfinding.LoadNavigationData(filePath,fileName);
          }
     }
```

```
	/*user interface code, allow you to load desired
navigation file by using simple interface, directly from play
mode
*/
	void OnGUI(){
	GUILayout.BeginVertical ();
	if(newFilePathDialog){
	GUILayout.BeginHorizontal ();
	GUILayout.Label("file path:");
	filePath =
GUILayout.TextField(filePath,GUILayout.MaxWidth(250));
	GUILayout.EndHorizontal ();
	GUILayout.BeginHorizontal ();
	GUILayout.Label("file name:",GUILayout.MaxWidth(65));
	fileName =
GUILayout.TextField(fileName,GUILayout.MaxWidth(100));
	GUILayout.EndHorizontal ();
		if(GUILayout.Button ("Load",GUILayout.MaxWidth(40))){
		Pathfinding.LoadNavigationData(filePath,fileName);
		newFilePathDialog = false;
			}
		}else{
		if(GUILayout.Button ("Load other file"))
		newFilePathDialog = true;
		}
	GUILayout.EndVertical ();
	}
}
```

In order to use this script, it's should be attached to the active game object in scene, after that, with help of little menu, you can quickly load navigation files in real time.

If you need to navigation file was automatically loaded by PathfindingManager inside compiled project, then do as follows:
- enter any folder name to a field “fileFolder” of PathfindingManager component, for example “NAPS_Data”;

- enter the name of the navigation file that must be loaded into field “fileName” of PathfindingManager component, for example “de_dust” and set autoLoad flag;

- compile the project;

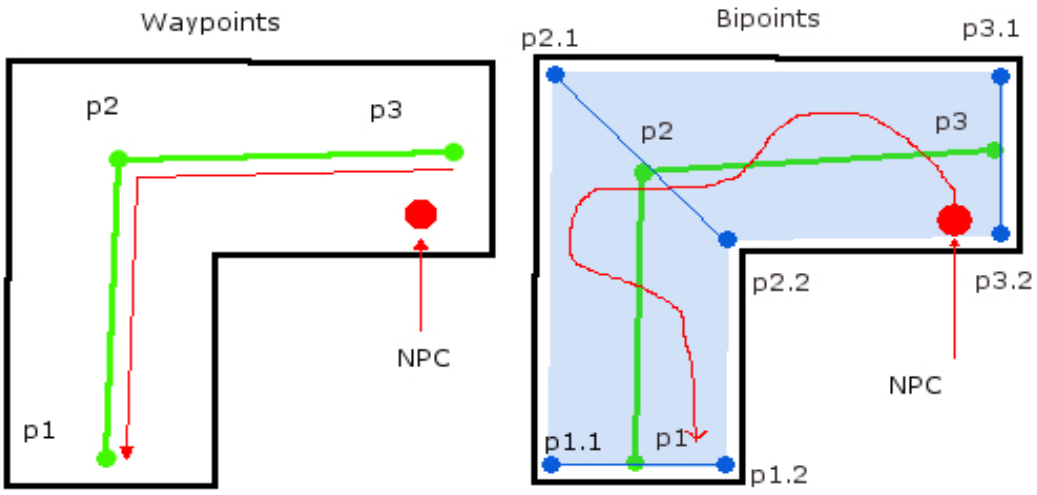
- create folder with name that you pointed above(“NAPS_Data”) inside *_Data folder of your compiled project and put to this folder navigation file(*.nvf) with name that you pointed above(“de_dust”), for example D:\CompiledProject\CompiledProject_Data\NAPS_Data\de_dust.nvf

From now after application with scene where is PathfindingManager attached to some active game object will be loaded, de_dust.nvf will be automatically loaded too.

What is bipoints and how to use them?

When I was developing a navigation system for our game initially as maximally simple and effective was selected solution based on waypoints. But the classic waypoints also have several serious drawbacks chief among which, for us, was that in fact the only path they describes is a line between two points. In our case we needed that the NPC could always change his moving trajectory during running through the same streets, rooms, etc, making path unique. Thus the problem in the case of classical waypoints can be solved by addition of new points, that mean increase in calculations and in some cases we had to add too many points to provide a unique way for NPCs. So I came up with the idea of modifying the classic single nodes - waypoints into another type of connection nodes, consisting of two points in space, i called them bipoints. Sense of bipoints is that between two points describes not just a line but the whole area which can be used by NPC to move through it. With this there is not complication for the algorithm of paths calculations and calculates pair of bipoints as two single points in space, amount of stored data remains almost the same.

Consider the example:



On the pictures we see an example of navigation for the same room. Black line shows the boundaries of the premises. On the picture «Waypoints» see the path laid indoors which consists of three points p1, p2, p3. The trajectory of any movement of the NPC by this way despite the large available space will be reduced to a broken line which are presented by connected waypoints, such as the trajectory of the NPC from point P3 to P1 (highlighted in red). It would be logical to apply smoothing when turns to the next point, but trajectory in fact will be the same. Of course to implement a more or less realistic behavior of the NPC we need more freedom of movement on the ground.

Now consider an example of the same navigation map but with using bipoints (picture «Bipoints»). In this example the path is calculated also by three points in the matrix of path-finding library of NAPS (Pathfinding.cs) while also contains information only about three points but in a scene each of the points p1, p2, p3 is presented by a two points (p1 - p1.1, p1.2; p2 - p2.1, p2.2; p3 – p3.1, p3.2), the only information about these points is have to store - it is their position in space. And so bipoint – is a node of path which describes a line in space, not just a point, but when calculating is only takes into account the central point of this line. In this case by connecting a pair of bipoints we've got an area described by 4 points, not by 2. Constructing a path with bipoints you get an area for the movement limited by the figure described by bipoints. Thus NPC can build any trajectory of movement within this area. The example of trajectory shown as red line in this case would come up for creatures such as zombies or some wandering animal, etc. But you can also build the most optimized way taking into account the angle of the cut, as far as possible for smart NPC using the extreme points of the available area as well as many other features. All functions for the construction of any types of movements are in the path-finding library of NAPS (Pathfinding.cs). For information about these features and their usage see "The functions of path-finding library to work with AI" and the accompanying examples of the movement AI.

The functions of path-finding library to work with AI

LoadNavigationData – loads desired navigation file. As parameters takes file path and file name without extension. File path and file name must be type of string. Number of overloads — 1.

public static void LoadNavigationData(string filePath, string fileName)

Пример(перегрузка №1):

```
public string fileFolder = “NAPS/NavigationData”;
public string fileName = “de_dust”;

void Awake(){
string filePath = Application.dataPath+"/"+fileFolder;
Pathfinding.LoadNavigationData(filePath,fileName);
}
```

DataLoaded – checks whether navigation data was loaded, if loaded – returns true, otherwise – false. Number of overloads — 1.

public static bool DataLoaded()

Пример(перегрузка №1):

```
void Awake(){
if(!Pathfinding.DataLoaded())
Debug.Log(“Navigation data was not loaded!”);
}
```

FindPathOfType – takes as a parameter the name of subnet of paths, such as string, returns the index of the desired subnet in an array Pathfiding.Paths. Number of overloads - 1.

public static int FindPathOfType(string type)

Example (overload №1):

```
int pathIndex = FindPathOfType(«air»);
Debug.Log(“Size path of type air =”+Paths[pathIndex].size);
```

GetWaypointInSpace — returns a point in the space of a single node or a specific point inside bipoint. If a single node, the parameter v is ignored and the result is a node position in space. If the node is bipoint, then with the aid of the values of v can be get any point inside bipoint, using values from 0 to 1, with v = 0 - the first point of bipoint, v = 1 - the second point of bipoint, v = 0.5 – center of bipoint. Number of overloads - 2.

public static Vector3 GetWaypointInSpace(float v, int index, string pathType)
public static Vector3 GetWaypointInSpace(float v, int index, int pathIndex)
public static Vector3 GetWaypointInSpace(float v, Node node)

Example(overload №1):

```
Node node = new Node(1, «air»);

void Start(){
Vector3 bipointCenter =
GetWaypointInSpace(0.5f,node.number,node.type);
Debug.Log(“Center of bipoint:”+bipointCenter);
}
```

Example(overload №2):

```
Node node = new Node(1, «air»);
int pathIndex;

void Start(){
pathIndex = FindPathOfType(node.type);
Vector3 bipointCenter =
GetWaypointInSpace(0.5f,node.number,pathIndex);
Debug.Log(“Center of bipoint:”+bipointCenter);
}
```

Example(overload №3):

```
Node node = new Node(1, «air»);

void Start(){
Vector3 bipointCenter = GetWaypointInSpace(0.5f,node);
Debug.Log(“Center of bipoint:”+bipointCenter);
}
```

GetPointTransform – returns a transform of specified node. Number of overloads – 1.

public static Transform GetPointTransform(int nodeNumber, string type)

Example(overload №1):

```
Node node = new Node(1, «air»);

void Start(){
Transform nodeTransform =
GetPointTransform(node.number,node.type);
if(nodeTransform.childCount>0)
Debug.Log(“Node is bipoint!”);
else
Debug.Log(“Node is single point!);
```

```
}
```

NodeInArray – checks if the desired node in the array of nodes, and if so, it returns true, otherwise - false. Number of overloads - 1.

public static bool NodeInArray(Node node, Node[] nodes)

Example(overload №1):

```
Node node = new Node(1, «air»);
Node[] array = new Node[2];

void Start(){
array[0] = new Node(Random.Range(0,10), «air»);
array[1] = node;
if(NodeInArray(node,array))
Debug.Log(“This node in array!”);
}
```

LinkExist – checks whether there is a link between two nodes, if so, it returns true, otherwise - false. Number of overloads - 1.

public static bool LinkExist(Node first, Node second)

Example(overload №1):

```
Node firstNode  = new Node(1, «air»);
Node secondNode = newNode(2, «land»);

void Start(){
if(LinkExist(firstNode,secondNode))
Debug.Log(“I can land on the
node :”+secondNode.number+secondNode.type);
}
```

FindClosestPointTo – finds the node, closest to the current position, from all visible nodes, of available for this AI, subnet of paths. As parameters, takes an array of available for this type of AI subnet of paths and position of NPC in space. Number of overloads - 1.

public static Node FindClosestPointTo(string[] types, Vector3 v3)
public static Node FindClosestPointTo(string[] types, Vector3 v3, float radius, int layerMask)

Example(overload №1):

```
string[] availablePaths = new string[2];
Node closestNode;

void Start(){
availablePaths[0] = “land”;
availablePaths[1] = “air”;
closestNode =
FindClosestPointTo(availablePaths,transform.position);
Debug.Log(“Closest node to this
position :”+closestNode.number+closestNode.type);
}
```

Example(overload №2):

```
string[] availablePaths = new string[2];
Node closestNode;
int ignorePlayerLayer;
float NPCradius;

void Start(){
ignorePlayerLayer = LayerMask.NameToLayer(“Player”);
ignorePlayerLayer = 1<<ignorePlayerLayer;
ignorePlayerLayer = ~ ignorePlayerLayer;
CapsuleCollider thisCollider = GetComponent<CapsuleCollider>();
NPCradius = thisCollider.radius;
availablePaths[0] = “land”;
availablePaths[1] = “air”;
closestNode =
FindClosestPointTo(availablePaths,transform.position,NPCradius,
ignorePlayerLayer);
Debug.Log(“Closest node to this
position :”+closestNode.number+closestNode.type);
}
```

GetRouteForPoint – returns route for desired point in space. Number of overloads -1.

public static RouteData GetRouteForPoint(RouteData newRouteData,string[] types,Vector3 destPoint,Vector3 plrPos,float plrRadius,int layerMsk)

Example(overload №1):

```
string[] availablePaths ;
RouteData routeData = new RouteData();
```

```
Vector3 destinationPoint;
int ignorePlayerLayer;
float NPCradius;

void Start(){
movement = GetComponent<NPCMovement>();
ignorePlayerLayer = LayerMask.NameToLayer("Player");
ignorePlayerLayer = 1<<ignorePlayerLayer;
ignorePlayerLayer = ~ ignorePlayerLayer;
CapsuleCollider thisCollider = GetComponent<CapsuleCollider>();
NPCradius = thisCollider.radius;
availablePaths = new string[Pathfinding.Paths.Length];
int i;
for(i=0;i<availablePaths.Length;i++){
availablePaths[i] = Pathfinding.Paths[i].type;
}
destinationPoint = GetRandomNavigationPoint(availablePaths);
if(destinationPoint !=Vector3.zero){
routeData =
GetRouteForPoint(routeData,availablePaths,destinationPoint,tran
sform.position,NPCradius,ignorePlayerLayer);
}

}
```

ResumeRoute – resumes route in case of lost visibility of current node. Number of overloads-1.

public static RouteData ResumeRoute(RouteData newRouteData,string[] types,Vector3 plrPos,float plrRadius,int layerMsk)

GetRandomNavigationPoint – returns random point in space, that is generated inside one of available subnet of paths. Number of overloads -1.

public static Vector3 GetRandomNavigationPoint(string[] types)

Example(overload №1):

```
string[] availablePaths ;
Vector3 randomPoint;

void Start(){
availablePaths = new string[Pathfinding.Paths.Length];
int i;
```

```
for(i=0;i<availablePaths.Length;i++){
availablePaths[i] = Pathfinding.Paths[i].type;
}
randomPoint = GetRandomNavigationPoint(availablePaths);
Debug.Log("random navigation point:"+randomPoint);
}
```

GetRandomPathsNode – returns a random node of one of the available subnets of paths. As parameter, takes an array of names of available subnets of paths. Number of overloads - 1.

public static Node GetRandomPathsNode(string[] types)

Example(overload №1):

```
string[] availablePaths ;
Node randomNode;

void Start(){
availablePaths = new string[Pathfinding.Paths.Length];
int i;
for(i=0;i<availablePaths.Length;i++){
availablePaths[i] = Pathfinding.Paths[i].type;
}
randomNode = GetRandomPathsNode(availablePaths);
Debug.Log("New random
node :"+closestNode.number+closestNode.type);
}
```

NodeIsEmpty – checks whether a desired node is exist, if not - returns true, otherwise - false. Number of overloads - 1.

public static bool NodeIsEmpty(Node node)

Example(overload №1):

```
Node node ;

void Start(){
node = new Node(-1, "");
//return true,because node is empty
//Debug will print : "Node exist : false");
Debug.Log("Node exist:"+!NodeIsEmpty(node));
}
```

NodeIsBipoint – checks whether the node is bipoint, if node is bipoint – returns true, otherwise - false. Number of overloads — 1.

public static bool NodeIsBipoint(Node node)

Example(overload №1):

```
Node node ;
string[] availablePaths = {“path1”,”path2”};

void Start(){
node = GetRandomPathsNode(availablePaths);
if(NodeIsBipoint(node))
Debug.Log(“node “+node.number+node.type+” is bipoint”);
}
```

NodesEqual – checks whether the nodes are equal, that is, if the numbers and types of nodes are the same, it returns true, otherwise - false. Number of overloads - 1.

public static bool NodesEqual(Node a, Node b)

Example(overload №1):

```
Node aNode = new Node (5, “human”);
Node bNode = new Node (6, “human”);
bool result;

void Start(){
result = NodesEqual(aNode,bNode);
//will print “false”
Debug.Log(result);
aNode = bNode;
//will print “true”
Debug.Log(result);
}
```

GetCurve – returns a curve between two points, takes as parameters the first and second node of type Node. Number of overloads - 1.

public static CurveData GetCurve(Node first, Node second)

Example(overload №1):

```
Node firstNode = new Node (2, “air”);
Node secondNode = new Node (6, “land”);
CurveData curveData;

void Start(){
if(CurveExistBetween(firstNode,secondNode))
curveData = GetCurve(firstNode,secondNode);
}
```

CurveExistBetween – checks whether there is a curve between two nodes, if so, it returns true, otherwise - false. Number of overloads - 1.

public static bool CurveExistBetween(Node first, Node second)

Example(overload №1):

```
Node firstNode = new Node (2, “air”);
Node secondNode = new Node (6, “land”);
CurveData curveData;

void Start(){
if(CurveExistBetween(firstNode,secondNode))
curveData = GetCurve(firstNode,secondNode);
}
```

GetCurveLength – returns the length of a certain curve. To get the length of the curve is used partition it into segments, the greater the number of segments, the more time spent on calculating the length. Number of overloads - 1.

public static float GetCurveLength(int segments, CurveData curve)

Example(overloading №1):

```
Node firstNode = new Node (2, “air”);
Node secondNode = new Node (6, “land”);
CurveData curveData;
float curveLength = 0f;

void Start(){
if(CurveExistBetween(firstNode,secondNode)){
curveData = GetCurve(firstNode,secondNode);
     if(curveData !=null)
curveLength = GetCurveLength(100,curveData);
Debug.Log(“Curve length = “+curveLength);
```

```
}
}
```

GetBipointLength – if node is bipoint returns distance between its points. Number of overloads— 1.

public static float GetBipointLength(Node node)

Example(overload №1):

```
Node node = new Node (1, “path1”);

float bipointLength;

void Start(){
if(NodeIsBipoint(node)){
     bipointLength = GetBipointLength(node);
     Debug.Log(“node “+node.number+node.type+” is
bipoint,length:”+bipointLength);
}
}
```

MoveByCurve – used for motion along a certain curve. Works with class MotionCurve. For more information, see discription of class MotionCurve. Number of overloads - 1.

public static MotionCurve MoveByCurve(MotionCurve motionCurve, Vector3 curPos, float increment)

Also see example of motion along a curve — FollowCurve.

GetPointOnBezier – returns the point on the curve in depending of passed value (0 - the starting point, 1 - the end point of the curve, 0-1 - intermediate points of the curve). Number of overloads - 1.

public static Vector3 GetPointOnBezier(float t, CurveData newCurve)

Example(overload №1):

```
Node first = new Node (0, “land”);
Node second  = new Node(1, “land”);
CurveData curveData;
Vector3 middleCurvePoint;

void Start(){
```

```
if(CurveExistBetween(first,second)){
curveData = GetCurve(first,second);
if(curveData != null){
//Get middle point of curve
middleCurvePoint = GetPointOnBezier(0.5f,curveData);
}
}
}
```

DistanceToNode - returns the minimal distance between point in space and a certain node. Number of overloads - 1.

public static float DistanceToNode(Node thisNode, Vector3 curPos)

Example(overload №1):

```
Node node = new Node (0, “land”);
float distance = 0f;

void Update(){
distance = DistanceToNode(node, transform.position);
Debug.Log(“distance to node:”+distance);
}
```

VisibilityOfNode – determines the visibility of a certain node from a specific point. Returns values: - 1 - if the node is not visible, 0 - if the node is visible and is a single point, 1 - if the node is bipoint and see only one of its points, 2 - if the node is bipoint and see both his point. Number of overloads - 3.

public static int VisibilityOfNode(Node node, Vector3 curPos)
public static int VisibilityOfNode(int nodeNumber, int nodePathIndex,Vector3 curPos,int layerMsk)
public static int VisibilityOfNode(Node node,Vector3 curPos,float playerRadius,int layerMsk)

Example(overload №1):

```
int pointVisibility;
Node node  =  new Node(1, “land”);
Vector3 nextPoint  = Vector3.zero;

void Update(){
pointVisibility = VisibilityOfNode(node,transform.position);
if(pointVisibility<0){
```

```
Debug.Log(“Node is not visible!”);
}else if(pointVisibility<1){
nextPoint = GetWaypointInSpace(0.5f,node);
}else{
nextPoint = GetWaypointInSpace(Random.value,node);
}
MoveTo(nextPoint);
}
```

ObjectIsVisible – checks visibility of point in space, returns true if point visible, false – otherwise. Number of overloads - 4.

public static bool ObjectIsVisible(Vector3 objPos,Vector3 thisPos, float radius)
public static bool ObjectIsVisible(Vector3 objPos,Vector3 thisPos, float radius, int layerMsk)
public static bool ObjectIsVisible(Vector3 objPos,Vector3 thisPos,Vector3 lookDir,float FOV)
public static bool ObjectIsVisible(Vector3 objPos,Vector3 thisPos,Vector3 lookDir,float FOV,int layerMsk)

Example(overload №1):

```
float NPCradius;
public Transform object;

void Start(){
CapsuleCollider thisCollider = GetComponent<CapsuleCollider>();
NPCradius = thisCollider.radius;
if(ObjectIsVisible(object.position,transform.position,NPCradius
)
Debug.Log(“object “+object.name+” is visible”);
}
```

Example(overload №2):

```
float NPCradius;
public Transform object;
int ignorePlayerLayer;

void Start(){
ignorePlayerLayer = gameObject.layer;
ignorePlayerLayer = 1<<ignorePlayerLayer;
ignorePlayerLayer = ~ignorePlayerLayer;
CapsuleCollider thisCollider = GetComponent<CapsuleCollider>();
NPCradius = thisCollider.radius;
```

```
if(ObjectIsVisible(object.position,transform.position,NPCradius
,ignorePlayerLayer)
Debug.Log("object "+object.name+" is visible");
}
```

Example(overload №4):

```
public Transform object;
int ignorePlayerLayer;
float fieldOfView = 54f;

void Start(){
ignorePlayerLayer = gameObject.layer;
ignorePlayerLayer = 1<<ignorePlayerLayer;
ignorePlayerLayer = ~ignorePlayerLayer;
CapsuleCollider thisCollider = GetComponent<CapsuleCollider>();
NPCradius = thisCollider.radius;
if(ObjectIsVisible(object.position,transform.position,transform
.forward,fieldOfView,ignorePlayerLayer)
Debug.Log("object "+object.name+" is in my field of view!");
}
```

GetRoute – returns a global route between two nodes (returns a sequence of transition points between the individual subnet of paths) as an array of nodes of type Node. Number of overloads -1.

public static Node[] GetRoute(string[] types,Node startPoint,Node endPoint):Node[]

Example(overload №1):

```
Node[] globalRoute ;
Node startNode = new Node(1, «type1»);
Node endNode = new Node(27,  «type3»);
string[] allowedPaths = {"type1", "type2", "type3", "type4"};

void Start(){
globalRoute = GetRoute(allowedPaths, startNode, endNode);
}
```

For more information about using this function, see the attached examples of AI.

GetPointInArea – returns a random point in space inside the area described by two nodes. Depending on the parameter range, can be obtained a random point in the entire region, or to cut part of the area in which the point is

not will be generated. Normally this is used to generate a random point within the region for NPC, which moves in the area. Thus the generated point will always be ahead of NPC, which ensures a movement to a desired node. In this case, by using the range can expand or narrow the scope of generating points ahead of NPC. Number of overloads - 1.

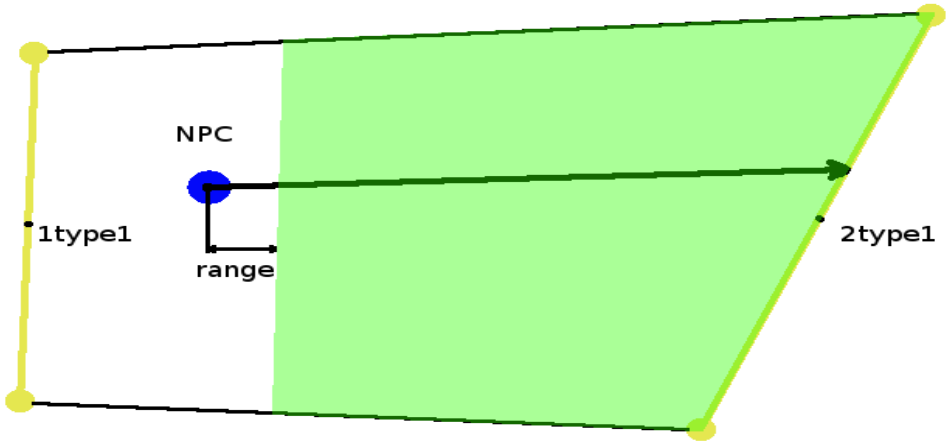
public static Vector3 GetPointInArea (Node first, Node second, Vector3 curPos,float range)

Example (overload№1):

```
Node startNode = new Node(0, “type1”);
Node endNode =  new Node(1, “type1”);
Vector3 targetPoint;

void Update(){
targetPoint =
GetPointInArea(startNode,endNode,transform.position,1f);
//MoveTo(targetPoint);
}
```

This example assumes that the NPC moves from node 1type1 to node 2 type1, transform.position - NPC position in the area described by these nodes. The scope for generating point (shown in green), in front of NPC, slightly reduced (about 1 m) with the parameter range.



When need to generate a random point within the entire region, described by two nodes, setting range must be set negative, e.g. -1.

GetOptimizedTrajectoryPoint – returns the value of a point inside the node as floating point value. This point is best to move to the next node after the current node, selection of the optimal point depends on the passed

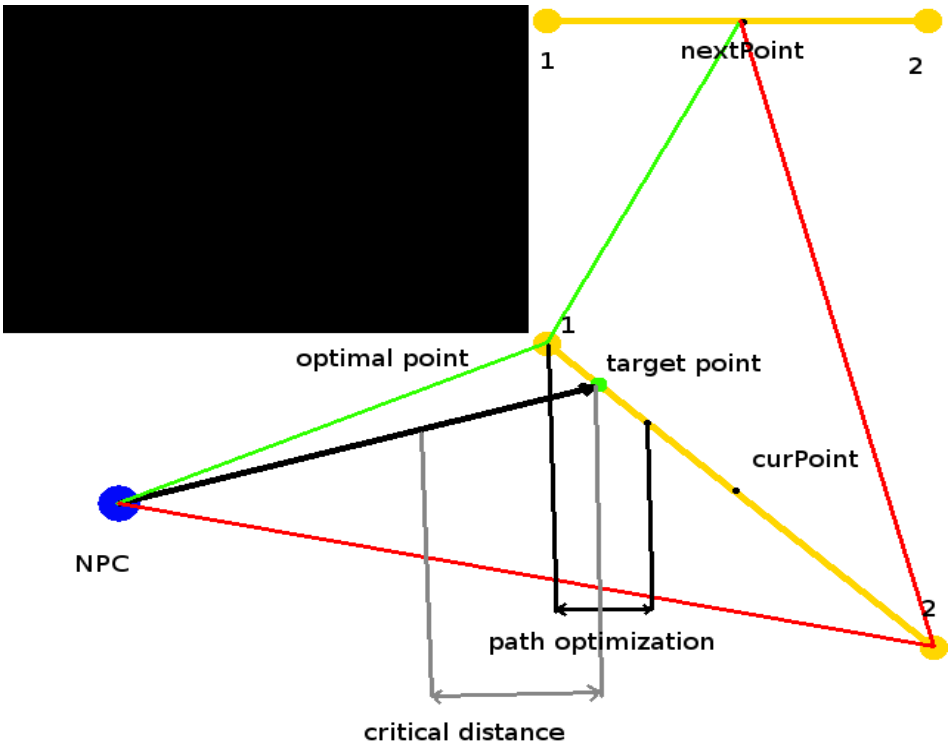
parameters. In order to convert the value to a point in space, should use the function GetWaypointInSpace. Number of overloads - 1.

public static float GetOptimizedTrajectoryPoint(Node curPoint, Node nextPoint, float lastValue, float pathOptimization,Vector3 curPos, float defaultAmplitude, float minDistance, float maxAmplitude):float

Example(overload №1):

```
Node curPoint = new Node(1, “type1”);
Node nextPoint = new Node(2, “type1”);
float pathOptimization = 2.3f;
float targetPointValue = -1f;
float criticalDistance = 7f;
Vector3 targetPoint;

void Update(){
targetPointValue = GetOptimizedTrajectoryPoint(curPoint,
nextPoint, targetPointValue, pathOptimization,
transform.position, Random.Range(0.5f,2f), 7f, 0.2f);
targetPoint = GetWaypointInSpace(targetPointValue,
curPoint.number, curPoint.type);
//MoveTo(targetPoint);
}
```



The picture above shows how the code works. In this example the NPC moves to the node nextPoint via the node curPoint (the current node). Nodes curPoint and nextPoint are bipoints that is composed of 2 points in space that are marked with numbers 1 and 2 at each node. In order to select the optimal point within a node curPoint to move to the node nextPoint the function determines: by which of two points of bipoint curPoint should move in order to minimize the length of the path. As you can see the path through the point 1 (shown in green) is more optimal than the path through the point 2 (shown in red). Thus the optimal point is the point 1 of bipoint curPoint and moving through this point will be most optimized. But this NPC have the parameters of optimization of path that do not allow him each time build too optimized path. This can be used for variability trajectory of NPC. One of the parameters of optimization of path are parameter pathOptimization, this parameter shows within what distance in meters from the optimal point can vary the real point of the movement. As can be seen on the picture the actual moving point (targetPoint), shown in green, located at some distance away from the optimum point but within parameter pathOptimization. If need to build the most optimized trajectory parameter pathOptimization for this NPC should be set to 0. Thus the real point for the movement will always be an optimal point. On the trajectory of the NPC are also influenced by the following parameters:
defaultAmplitude – this parameter determines the amplitude variations of the actual point within the parameter pathOptimization, if the distance to the real point is greater than the criticalDistance;

criticalDistance – parameter, which determines how far away from the real point, variation of amplitude of this point is reduced;

maxAmplitude – parameter, which is the amplitude of variation of real point within bipoint, when the parameter criticalDistance is reached;

In this example the NPC has the option to optimize the way pathOptimization which is equal to 2.3, this mean that after computing the optimal point for the movement through bipoint the real point will vary between 2.3m from the optimal point. The amplitude variations of the real point relative to the last value at a distance of more than 7 meters from this point will be from 0.5m to 2m and after reaching a critical distance (7m or less) the amplitude of variation will be equal 0.2 m from the last real point within the parameter of optimization.

Thus using this function the trajectory of the movement of NPC can be varied to make it more optimized or vice versa - to make the unpredictable trajectory.

The classes of path-finding library to work with AI

Class Node

To work with the navigation at your map you will have to constantly refer to the specific nodes of subnets of paths, search for the routes between them and more. For this purpose serves the class Node (Pathfinding.Node)

Node – the most widely used class to working with navigation, it is used to represent nodes.

Variables:

`public int number` – used to store number of node;
`public string type` – used to store type of node (type of subnet);

Usage example:

Suppose that we need to get the position in space of the node with the number 2 and type «road1», if the node is bipoint then get its central point. This can be done as follows:

```
/ * declare a new instance of the node, where -1 - initial
number, and "" - string representation of the type of the
node, the node type – is a variable of type string, you can
make up any name of type when creating paths in the editor.
The number -1, and empty quotes, instead of name of type, show
that this node is empty yet.
* /
Pathfinding.Node node = new Node(-1, “”);

void Start(){
/ * Now we need to set the required values to the empty node,
we can do it like this:
* /
node.number = 1; / * as the numbering is from 0, the real
value of a node in the code will be equals the number of node
in editor minus 1, that is 2-1 = 1.
* /
node.type = “road1”; //set the type to this node
/*declare a variable of type Vector3 and using the function
GetWaypointInSpace obtains the desired position within the
node;
```

```
*/
Vector3 nodePosition =
Pathfinding.GetWaypointInSpace(0.5f,node.number,node.type);
}
```

Class RouteData

To work with the routes used class RouteData. If you want to your AI was able to use paths to moving through them then you will need to create an instance of the class RouteData for this AI. Navigation in NAPS is represented by a grid which consist of nodes. Connections between the nodes represent different paths. Paths can be of the same type or be atypical. The path of the same type - a path that consists of the nodes of the same type. For example, such path will be of the same type: 0 "human" - 3 "human" - 6 "human". Atypical path - a path, passing through the nodes of different types. An example of an atypical path: 0 "human" - 2 "dragon" - 6 "human". Within the overall grid of paths nodes of one type constitute a subnet. Such a subnet will be pre-calculated and if the NPC move only within the only one subnet then he will not have to calculate the path for himself at all since it will just have to access the calculated subnet. In the case of NPC must be able to travel directly through multiple subnets formed by nodes of different types it is necessary to calculate the optimal path between different subnets of paths but using precalculated data of each subnet. This approach allows us many times to reduce the complexity of the calculation. And so, the class RouteData is essentially a repository of the found route, with the only difference is that it takes into account the precalculated data of subnets of paths as well as some ancillary data. Because of this each instance of AI does not store the entire route, i.e. each point of the ways in which it will move, but just only transition point between the precalculated subnets of paths an intermediate points in these areas are obtained dynamically from pre-calculated subnets. In most cases you do not have to remember all these details, it is enough to learn the examples and understand how to use this class.

Variables:

`public Node[] route` – stores transition points of the route, since – global route;

`public Node curPoint` – stores the current node in the route, where move;

`public Node nextPoint` – stores the next node in route, after current node;

`public Node destinationPoint` – stores destination point of current route;

`public int nextPointIndex` – stores the index of the transition point for the type of subnet of paths, which within at the moment is moving NPC. This

variable is used only for the correct calculation of the route, usually do not have any need to access it because its used only by functions of path-finding library.

For more insight see an example of using the class RouteData in applied example of AI of following by paths PathFollowingAI.cs.

Class MotionCurve

In case we need to organize a movement of any object along a curve you need to use a class MotionCurve. This class is used to store movement data for the current curve.

Variables:

`public CurveData curve` – current curve, in which the motion;
`public float curveLength` – the length of current curve;
`public float passedDist` – distance, passed by curve;
`public Vector3 lastPoint` – stores the last position of an object moving along a given curve (needed for correct calculation of distance passed by the curve);
`public Vector3 newPoint` – stores a point in space to which have to move, at this time, to follow the curve;

Almost all of the variables mentioned herein are used by functions of the movement along the curve and are of no interest unless you want to delve into the process of moving along a curve. In order to organize a movement along the curve without going into details it is enough to use a variable newPoint which always contains the point to which have to move to follow the curve. In order to understand how to use the class MotionCurve see an example of the movement along the curve FollowCurve.cs.

Class CurveData

This class is used to store and recording data of Bezier curves into a file. In NAPS used Bezier curves of third order, they are represented by 4 points in space where the first and last points - are respectively start and end points of the curve and the two intermediate points - is the control points that define the shape of the curve.

Variables:

`public Vector3S[] tangents` – array representing the points of the curve (tangents [0] - the starting point of the curve; tangents [1] - the first control point; tangents [2] - the second control point; tangents [3] - the end point of the curve.

Since the class CurveData is not only used for temporary storage of data of the curves but also to write it to a file, instead of the usual Vector3 used a different class - Vector3S. This class has only two functions that helps save the data to an instance of a class or download them from it:
public Vector3S(UnityEngine.Vector3 v3) – used to store a variable of type Vector3 into instance of class Vector3S on its declaration;
public UnityEngine.Vector3 GetVector3() - returns a Vector3 saved before;

Example: Let's define a Bezier curve of third order with the help of a class CurveData and a helper class Vector3S.

```
//create an instance of the class CurveData to store new curve
CurveData newCurve = new CurveData();

void Start(){
//set the size of array of curve points equal to 4
newCurve.tangents = new Pathfinding.Vector3S[4];
//sets the start point of the curve
newCurve.tangents[0] = new Vector3S(new Vector3(0,0,0));
//sets the first control point of the curve
newCurve.tangents[1] = new Vector3S(new Vector3(5,4,2));
//sets the second control point of the curve
newCurve.tangents[2] = new Vector3S(new Vector3(7,6,8));
//sets the end point of the curve
newCurve.tangents[3] = new Vector3S(new Vector3(10,0,10));

//now get back saved points, in form of Vector3
Debug.Log("first point of
curve:"+newCurve.tangents[0].GetVector3());
Debug.Log("control point
№1:"+newCurve.tangents[1].GetVector3());
Debug.Log("control point
№2:"+newCurve.tangents[2].GetVector3());
Debug.Log("end point of
curve:"+newCurve.tangents[3].GetVector3());
}
```

In most cases, you do not have to dive into the intricacies of using variables of this class, because for you it will do functions for working with curves.

Class Path

This class is used to store data of subnets of paths. Most of the variables of this class will not be used directly, as when working with navigation is not necessary.

Variables:

public string type – type of this subnet;

`public float[,] connectionLength` – the matrix of connections of nodes of this subnet;

`public float[,] pathLength` – the matrix of the lengths of all paths between all nodes of this subnet;

`public int[,] nextPoint` – the matrix of all routes between all nodes of this subnet;

`public int size` — number of nodes in this subnet;

`public Waypoint[] Waypoints` -an array of instances Waypoint class, which contains all nodes transforms of the subnet, and all atypical connections of each node of subnet.

When using functions of the library Pathfinding.cs usually it is not need to access the variables of the class. In most cases can only be useful variables size and type.

Base NAPS components

NAPS library of methods and classes - Pathfinding

Pathfinding library is a base of NAPS, it contains loaded navigation data as well as methods and classes to work with them. It also contains methods and classes that makes creation of AI more simplified. Calls to this methods and classes can be done through Pathfinding class (for example: Pathfinding.Node, Pathfinding.RouteData, Pathfinding.NodeIsBipoint()).

Navigator component

From NAPS version 1.3 all base navigation stuff placed in Navigator component. It's done for increasing universalization and make creation of AI more simpler. For now all what need to do in order to NPC will start moving, is attach to him NPCMovement, Navigator and pass destination point into Navigator component. All work about following by route to destination point Navigator will done himself and after that, it will pass desired point in space into NPCMovement component, in order to directly translate NPC to it. Navigator has a number of parameters which helps handle NPC movement.

Parameters:
destinationPoint – destination point that have to be reached;

distanceToDestinationPoint – straight-line distance to destination point (read only);

destinationPointIsVisible – contains true if destination point is visible, false – otherwise (read only);

distToCurPoint – distance to current point in space, since NPCMovement.targetPoint (read only);

Auto Ramble – is responsible for generation of random routes for NPC. If flag is set, then NPC will generate random destination points for himself inside navigation areas. In other case, i.e. if flag is not set, then NPC will be not move unless destination point was not set through Navigator.destinationPoint.

Move To Destination If Not Zero – defines whether the NPC can move to the destination point if it not equal to Vector3.zero. If flag was set, then NPC will start moving to destination point if it not equal to Vector3.zero and after NPC will reach destination point it will reset it to Vector3.zero. Destination point in this case have to be passed to Navigator through Navigator.destinationPoint variable.

Use Visibility - is responsible for the checking of visibility of path nodes in order to build a path more rationally. If flag was set, then NPC will go to the next node in route as only next node became visible. Otherwise if flag was not set, NPC have to reach a distance to current node in route, specified as pointReachRadius, in order to be able to move to next node.

Use Destination Point Visibility – defines whether NPC can take a destination point as reached in case of its visibility. If flag is set – he can, otherwise NPC have to reach distance specified as pointReachRadius in order to be able take destination point as reached.

Use Node Distance – determines in what way distance from the NPC to the current node must be measured. If flag is set, then distance takes as minimal distance between current node and NPC position, independently on to what point inside this node NPC actually moves. In other case, if flag was not set, distance measures as distance between NPC position and actual point inside this node(NPCMovement.targetPoint).

Note: second method works faster, so it can be more relevant for crowd simulation.

Point Reach Radius – distance that NPC have to reach in order to current node or destination point could be taken as reached, in case of not using visibility for this purpose.

Path Optimization – distance in meters, that shows maximal deviation from the optimal point inside current node while moving to the next node.

Trajectory Update Interval – interval of update actual point inside current node(measured in seconds).

Note: If you don't need of dynamic variety of trajectory of movement, then you should set this value equals to Mathf.Infinity.

Types – array of available for this NPC subnets of paths. Names must be type of string.

Modify target point coordinates - determines what coordinates of actual point NPC can get for movement. For example, if you want to ignore "y" coordinate of path nodes, then check-box "y" should be unchecked. Thus NPC will updates only 2D coordinates (x and z) from the current node.

If you need that NPC did not just ramble and go to desired point in space, then you have to do as follows:
- be sure that NPC has a NPCMovement component;
- be sure that NPC has a Navigator component;
- uncheck Auto Ramble check-box of Navigator componenet;
- check Move To Destination If Not Zero check-box;
- pass desired point in space as Vector3 into Navigator.destinationPoint;

In order to pass destination point into Navigator it is good way to do this through custom script, which will updates this point. Take a look at example script, which passes position of some object as destination point to the Navigator. Attach this script to NPC unit then drag and drop some GameObject from scene into its Target Object field. After you pressed "Play" button you can drag this object in scene view and you will see that NPC always goes to this object.
Script code:

```
using UnityEngine;
```

```
using System.Collections;

public class FollowObject : MonoBehaviour {
	//объект к которому нужно двигаться
	public Transform targetObject;
//расстояние между NPC и объектом при котором объект будет задавать
свою позицию как конечную точку
	public float distance = 1f;
	Navigator navigator;

	// Use this for initialization
	void Start () {
		navigator = GetComponent<Navigator>();
		if(navigator){
			//запрещаем генерацию случаного маршрута
			navigator.autoRamble = false;
//разрешаем движение к конечной точке, если она не равна
Vector3.zero
			navigator.moveToDestinationIfNotZero = true;
		}
	}

	// Update is called once per frame
	void Update () {
		if(navigator && targetObject){
if(Vector3.Distance(transform.position,targetObject.position)>
distance)
			navigator.destinationPoint = targetObject.position;
		}
	}
}
```

In this example destination point sets up through Navigator.destinationPoint variable, with this flag moveToDestinationIfNotZero is set and this makes NPC go to the destination point. But you can make NPC go to the destination point by call MoveToDestinatination(Vector3 destPos) method of Navigator from Update() void of any active script, but for this purpose autoRamble and moveToDestinationIfNotZero must be setted up to false.

Пример:

```
void Update(){
Navigator navigator = GetComponent<Navigator>();
if(navigator)
navigator.MoveToDestination(new Vector3(0,20,10));
}
```

The difference between this methods is only in that the first method automatically resets destination point after it was reached(makes it equal to Vector3.zero), while the second method is just updates data about distance to the destination point and its visibility at the moment, but not reset it automatically. So second method may be useful in case when you don't need to reset destination point automatically and only need to get information whether the destination point was reached, maybe for modify its coordinates or something like that.

Note: should remember that passed destination point must always be able (be visible) for at least one node of at least one of available for this NPC subnets of paths.

In this version of NAPS used layer-based visibility checkings. For this purpose uses two layers: first layer – used for game objects/NPCs that uses navigation, objects on this layer is not prevents to visibility checkings of each other; second layer – used for other dynamic game objects, that have to not prevents of visibility checkings of the path nodes, but prevents visibility checkings of the objects on first layer. For example “Player” - layer on which placed all NPCs/players. “Dynamic Object” - layer on which placed other dynamic objects (dynamic barrels, tables, doors etc.). Let's presume that we have a game level which have a couple buildings, each of which has at least one door that provide entering/exiting from this building. That door is controlled by some script and can be opened or closed by some command from this script. So we need to our NPCs could use this doors. In this case it is required to put all doors on “Dynamic Object” layer, then they will not prevents to visibility checkings of path nodes and NPC will follow to desired path node placed behind the door just like door is not exist on its path. So it will follow until it came up too close to the door and with help of some method of obstacles detection it will know that this obstacle – controllable door and it can be opened by some command. At same time NPCs on the layer “Player” located inside building will be not visible through the door for that NPC until it opened that door. Also using layer “Player” may be useful when need to check visibility of some NPC ignoring other NPCs or remove collision detection between NPCs.

Thus for using Navigator component it is necessary to reserve two layers for NPCs and other dynamic objects respectively and enter its numbers to the fields Player layer and Object layer of NPCMovement component of each NPC.

If you don't use dynamic objects in your game, and you need only player layer for your purposes, then enter to the field Object layer same number as in Player layer. **Don't forget to put your players/NPCs to player layer and dynamic objects to dynamic object layer.**

Note: *Should remember that visibility checkings uses raycasting for each NPC unit, so in crowds simulations performance can be seriously dropped down. Same thing with methods of obstacles detection and character grounding of NPCMovement component. So for crowd simulation for increasing performance it is better to disable that things (Navigator.useVisibility = false, Navigator.useDestinationPointVisibility = false,Navigator.useNodeDistance = false, NPCMovement.moveOnlyIfGrounded = false, NPCMovement.checkObstaclesInForward = false).*

NPCMovement component

NPCMovement component is responsible for object movement directly to target point in space taking into account acceleration, rotation etc. This component can be used for translation any object in scene, but first of all it's oriented to NPC movement.

NPCMovement properties:
targetPoint – target point in space, where need to go;

lookAtPoint – point in space used for rotation forward axis of this object into its direction;

grounded – shows whether the object is located on some surface, if flag Move Only If Grounded is not set, then always returns true.

Max Speed – maximum speed of object movement.

Acceleration Time - acceleration time in seconds.

Deceleration Time – deceleration time in seconds.

Movement Smooth – coefficient of smoothness of changing movement direction(less value – more smoothly changing).

Movement Smooth Distance – distance from target point at which proceeds interpolation of movement direction. This parameter helps to prevent overshoot of target point when object moves too fast, due to this reserve distance movement point interpolates correctly (in most cases this value is not require to be changed, even for fast moving objects value 15 is enough).

Rotation Speed – rotation speed of the object.

Auto Rotation Speed – whether the object change rotation directly with movement direction changing (uses if Rotate To Look Dir flag is not set).

Rotate Y Only – determines whether the object can rotates only around its Y axis.

Rotate To Look Dir – determines whether to rotate object in direction defined by lookAtPoint variable. i.e. if you need that the object rotates by its forward axis to lookAtPoint, then you need to pass desired point in space into NPCMovement.lookAtPoint and set Rotate To Look Dir flag. In other case object will rotates by its forward axis to the movement direction.

Can Move – if flag is set, then object can moves, otherwise movement disabled.

Can Rotate – if flag is set, then object can rotates in desired direction, in other case this object will be not rotated by NPCMovement.

Use Slerp – if flag is set, then movement vector will be interpolated spherically, otherwise – linearly.

Move Only If Grounded – if flag is set, then object will be able to moves only if it's grounded.

Check Obstacles In Forward – if flag is set, then object/NPC will be check closest space for obstacles and if obstacle detected, then movement will be skipped.

Player Layer – number of player layer.

Object Layer – number of dynamic obstacles layer.

Contact information

If you have a problem that can not be solved with the help of this guide, you have some information about errors in the work, or you just have a question related to NAPS, you can ask your question at: NavigationAndPathfindingSystem@gmail.com, author - Vladimir Maevskiy.