User's manual for "PathFinder 3D"

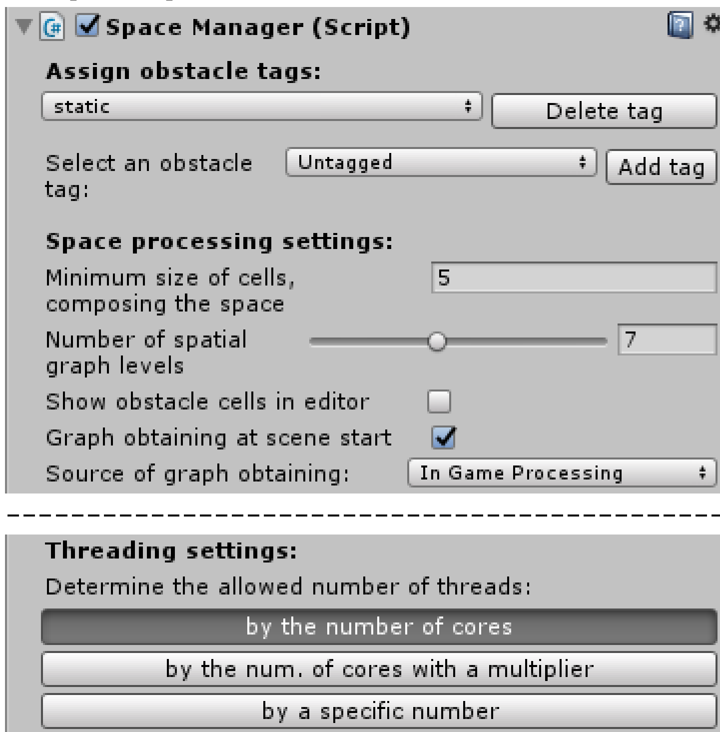
Version 0.3.4

Patch notes v0.3.4

Unfortunately, in this update we've not realized any changes that affects the functionality of the asset. On the other hand, we carried out work that accelerated the processing of obstacles and made the use of multithreading more efficient.

So, here is a list of all the changes:

1. We've implemented non-blocking dictionary, and now it is used for storaging occupied cells. So, now access to cells from different threads is faster.
2. Obstacle handling distribution to async tasks was improved. This sped up handling process.
3. Configuring the use of multithreading is now available in the SpaceManager script inspector.



4. Fixed a bug due to which the increment of the counter of processed triangles did not occur correctly.
5. Added XML comments to the following classes: SpaceGraph, SpaceManager, SpaceHandler, Pursuer.

General info about the work of Asset PathFinder 3D (v0.3.4).

Our Asset "PathFinder 3D" is designed to find the actual path (trajectory) in the space between two points. The found path will be the closest to the minimum possible and in most cases will not cross obstacles.

Pathfinding is performed using one of the following algorithms:

1. Modification of the algorithm A * with weighted heuristics;
2. Wave algorithm (the Lee algorithm).

From now and on, game objects that use the possibilities of searching the path and following it will be called "*Pursuers*".

The search space is discrete and consists of the cells that form the search graph. Each cell can be passable, or impassable, just this characteristic determines the possibility of finding a path through it.

In order to make it possible to find a way, it is necessary to explore the game space for obstacles, i.e. build the search graph. The processing of the game space must be executed once, before any of the pursuers want to look for the path (we recommend doing this at the start of the scene). If the game scene changes during the game (for example, obstacles move or resize and rotate), you can recalculate the search graph in the neighborhood of those obstacles that have changed. However, you should not do this too often, because the process of handling obstacles is laborious enough.

All entire functionality of the Asset is contained in two MonoBehaviour Classes: *SpaceManager* and *Pursuer*.

The Pursuer Class can be found in the /PathFinder3D/Scripts/Controllers folder. The Pursuer is designed to find the path and follow it. All game objects, for which the ability to search for and follow the path (pursuers) is needed, should have the Pursuer.cs script among its components.

There is a possible scenario, when there will be many pursuers on the stage. In this case, the procedure for finding the path will be called quite often, which can lead to a drop in performance. For this purpose, the pursuer's queue is organized in the SpaceManager Class. The pursuer in the queue are given permission to search the path, according to the number of pursuers for which the path is computed at a time that does not exceed the number of logical cores of the processor.

The SpaceManager Class can be found in the /PathFinder3D/Scripts/SpaceProcessing folder. SpaceManager is designed for scene processing and obstacle control, so it manages the queue of pursuers. The SpaceManager script must be initialized just once and be present on the game scene in a single instance.

We will consider the obstacles as game objects that satisfy the following conditions:

- The object has an enabled MeshCollider with the assigned mesh;
- There is an enabled MeshFilter with the assigned mesh on the object, and the gaming object has a forbidden tag;
- The object has an enabled Terrain.

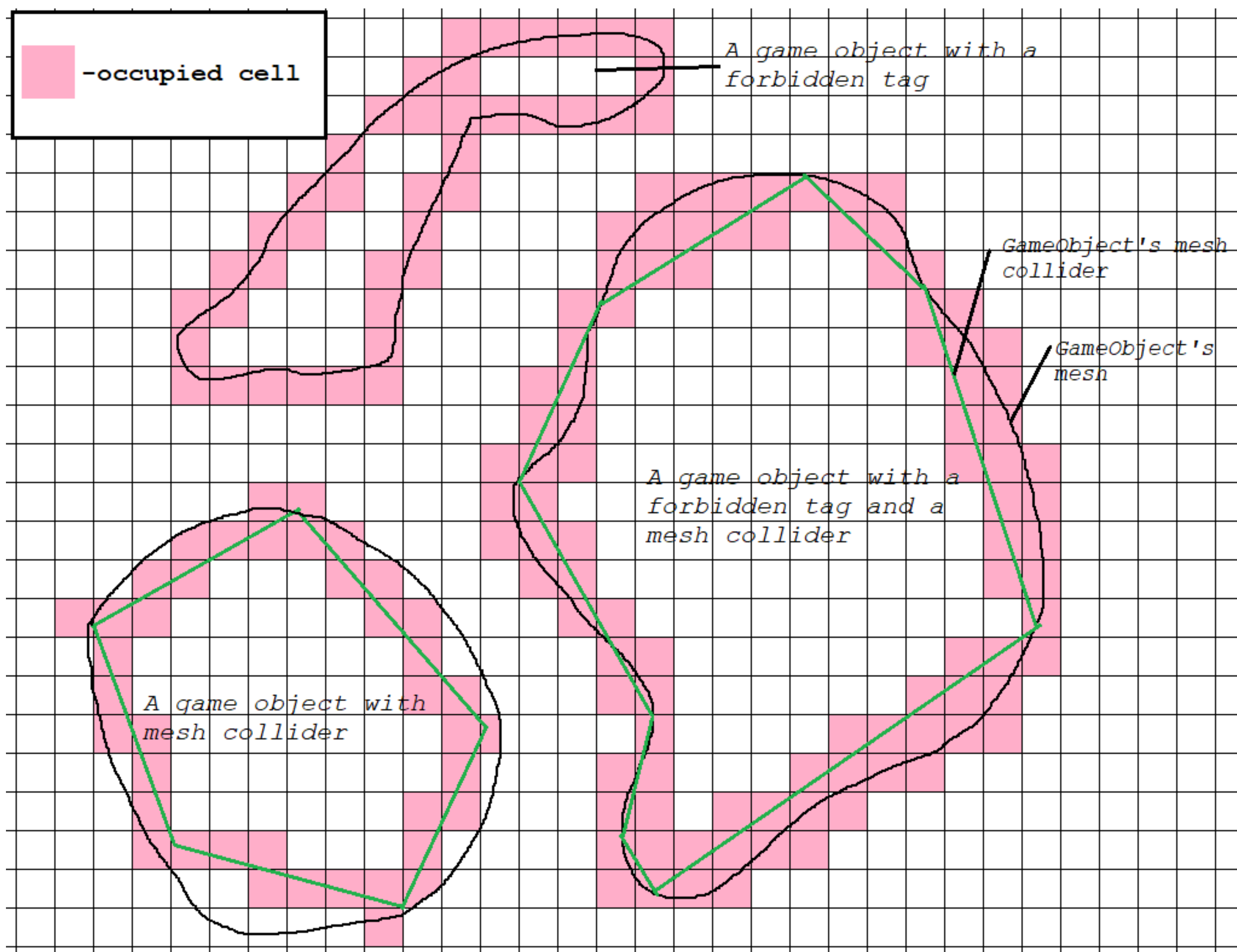
All objects that have a Pursuer component will not be considered as an obstacle.

By processing a scene, game objects with forbidden tags will be equated to obstacles. The calculation of the cells occupied by such objects will be based on the Mesh assigned to the corresponding field of the MeshFilter component.

By processing obstacles, the choice of occupied cells will be based on the individual components of the game object. Therefore, if the game object contains several components inherent in the obstacle (for example, Terrain and MeshCollider), then the selection of the occupied cells will be performed for each component.

Unfortunately, at the current moment, the possibility of processing obstacles equipped with the following components is not implemented: BoxCollider, SphereCollider, CapsuleCollider. But such obstacles can be handled based on the Mesh of the component selected in the MeshFilter, if the obstacle game object has a forbidden tag.

Schematically, the result of processing several obstacles will look like the following (two-dimensional projection):

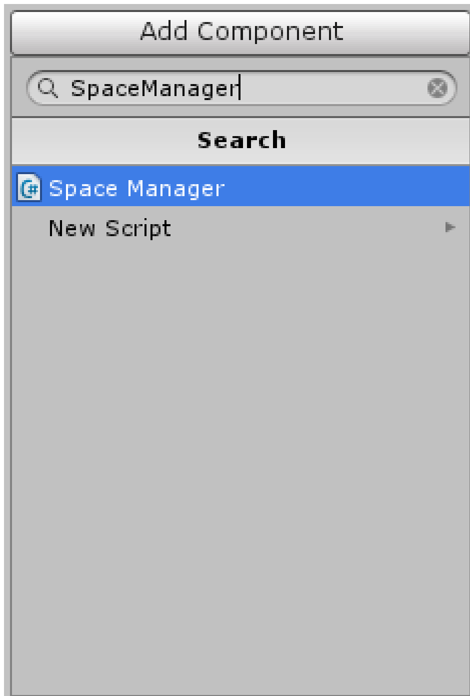


The usage of Asset

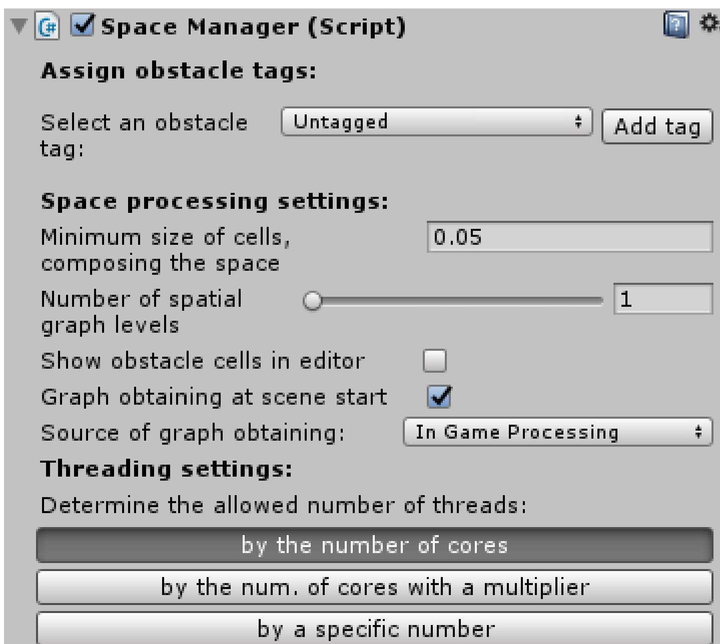
1. In the play scene, assuming that the particular asset is being used, you have to add the MeshCollider to all of the given objects, which can't be passed through (except Terrain). For other impassable objects, assign forbidden tags.

2. Create an empty game object and place the "SpaceManager" script on it.

The script "SpaceManager" should be present on the stage in a single copy.



3. Configure the added script.



Consider the blocks of parameters.

The ***"Assign obstacle tags"*** block allows adding and removing tags to the list of obstacle tags (prohibited tags).

The ***"Space processing settings"*** block provides the following settings:

- You can specify the minimum size of cells to which space will be divided;
- Setting the number of layers in the search graph (each higher level of the search graph consists of cells whose size is equal to the size of the cells of the previous level + 0.33 of the minimum cell size, ie cells of two neighboring levels differ in size in 0.33f * cellMinSize);
- Turn on / off the display of occupied cells in the scene editor window
- Enabling / disabling obtaining of a graph when the scene starts
- Switching the method of obtaining the graph. This can be a calculation at the start of a scene ("In Game Processing"), or loading a graph from a binary file, where it was previously serialized ("Desrializing From File") (Serialization and writing to a file can be done using the button from the image below).

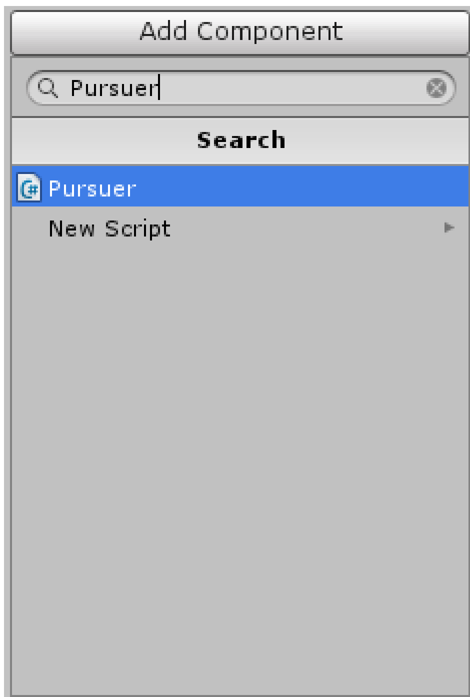
The number of levels of detail should be selected experimentally. Typically, the optimal number is 5. The more levels of detail you put, the longer the process of processing space will last.

Warning! The size of the cells should not be too small. Reducing the size of cells leads to an increase in their number and, as a result, slowing down all calculations and increasing the load on the CPU. The size should be well-thought-out.

The ***"Threading settings"*** block provides settings for using multi-threading, which are used when handling obstacles, as well as when searching for a path. Here you can choose the method by which the limit on the maximum number of worker threads will be set. The following describes the result of selecting one of the three available methods for limiting the number of threads.

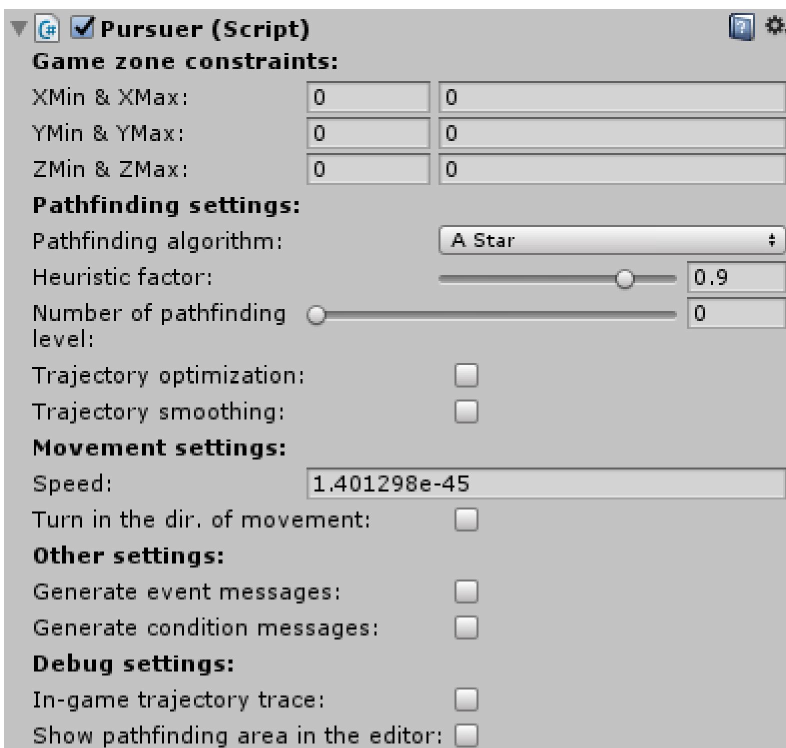
by the number of cores	The maximum number of threads will be equal to the number of software cores of the platform.
by the num. of cores with a multiplier	The maximum number of threads will be equal to the number of software cores of the platform, multiplied by the factor that you can set yourself. This is useful if you want to use for example half of all available cores.
by a specific number	The maximum number of threads will be equal to the number you specify.

4. Put the script called "Pursuer" on all of the game objects-pursuers



5. Configure the script.

You will see the following list of parameters:



Let us explain their meaning.

The first **"Game zone constraints"** parameter block determines the position and dimensions of the zone (the parallelepiped in world coordinates) in which the path can be searched.

XMin,XMax – restrictions on the X axis, (similarly to y, z).

*The second parameter block "**Pathfinding Settings**"* determines the used algorithm of the pathfinding and the final view of the path.

Also, there is a choice of the level of the spatial graph on which the search will be performed (Number of pathfinding level - slider). The choice of level essentially determines the size of the cells on which the path will be searched. The level number can be set from 0 to the number of levels selected in the SpaceManager.

The algorithm A * should be chosen if the scene contains many open spaces, and the obstacles on it take up less space than the free space.

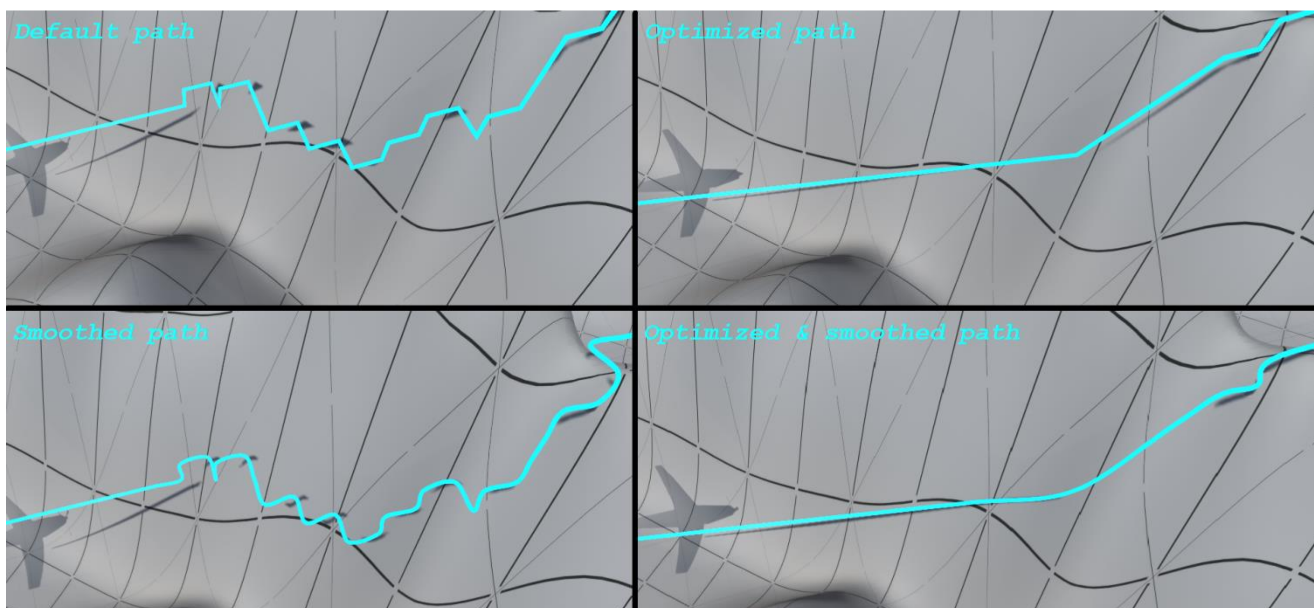
The wave algorithm should be selected if your scene is sufficiently "cramped". For example, it is a closed labyrinth, or a system of tunnels. It is best to choose an algorithm by performing tests on the stage.

When the algorithm A * is selected, the parameter "**Heuristic Factor**" becomes available. It allows you to choose between the minimum path and the search time. The acceptable range (0.5f, 1f). 0.5f is the shortest path possible. 1f is the fastest search time. The recommended value is -0.9f.

The parameter "**Trajectory Optimization**" is responsible for optimizing the trajectory. If it has a true value, then the found trajectory will minimize the surplus and simplified as much as possible.

The parameter "**Trajectory Smoothing**" allows you to enable smoothing of the found trajectory. The points of the smoothed trajectory will always be in a sufficiently small neighborhood of the points of the original trajectory. Be careful when using smoothing. A slight deviation from the original path is inevitable when interpolating the original path. Because of this, the smoothed trajectory can touch the surrounding obstacles in some places.

The meaning of these parameters is reflected in the picture below:



*The third parameter block "**Movement settings**"* determines the characteristics of the object's motion along the constructed path.

It allows you to adjust the speed of movement (in Unity coordinates) along the found trajectory, as well as to turn on / off the turn of the pursuer towards the motion

vector (whether the local direction vector (z) of the object is co-directed with the motion vector).

*The fourth block **"Other settings"*** allows you to enable or disable the sending of messages to the game object. For further information about messages, see "Pursuer Class states and events".

*Block **"Debug Settings"***, allows you to draw a found trajectory.

6. After processing the scene, call the MoveTo () function from the Pursuer class.

If you set the parameter "Process At Startup" when setting up SpaceManager, then the processing of the scene will begin at the moment the scene starts. Upon completion of the scene processing, all the game objects containing Pursuer among their components will receive a message "**TheGraphIsReady**". Receiving this message should be regarded as a signal about the readiness of the search graph. Call MoveTo() only after the scene has been processed. Otherwise exception "**GraphNotReadyException**" will be thrown.

The MoveTo() function takes one parameter of type "Vector3" or "Transform", meaning the target to move to. After the function is called, the search for the path to the destination begins. At the end of the search, the object will start moving to the target on the found trajectory.

An example of using MoveTo() after processing a scene:

Let the game scene have a configured SpaceManager with the parameter "Process At Startup" set to the true value, as well as a "Pursuer" with Pursuer and PursuerController scripts as components. Then following the target can be started as follows:

PursuerController.cs

```
using UnityEngine;

[RequireComponent(typeof(Pursuer))]
public class PursuerController : MonoBehaviour {
    public Transform target;
    //TheGrapIsReady() will be executed when the gameObject receives the 
"TheGraphIsReady" message
    public void TheGraphIsReady()
    {
        gameObject.GetComponent<Pursuer>().MoveTo(target);
    }
}
```

To abort the following to the target and put Pursuer in its original state, use the ResetCondition() method. Calling this method is possible at any time and at any state of the pursuer. The call will move the pursuer to its original state, stopping all active threads.

So, here is a list of rules for successful use of the Asset:

-the Pursuer and Target must be inside the search area configured in the Pursuer script.

-for the start of the path search, the space around the pursuer must be free of obstacles in a radius of at least the Cell Size set in the SpaceManager script. The same goes for the target of the pursuit.

-the size of the cells in the space that is configured in SpaceManager must be well-thought-out. (Do not do it if it's unnecessary)

-at the moment when the path search starts, the scene must undergo the initial processing of the obstacles or loading graph from a binary file. (SpaceManager.isPrimaryProcessingCompleted == true)

-the Pursuer should not be inside impassable objects, the same is true for the Target of pursuit.

This is all you need to know as a new user of our Asset.

But we have a lot of opportunities still in stock, and if needed, you can read about them below.

We will be very grateful if you leave a detailed review of our asset on the asset store page, based on experience of use, of course.

The Additional Features

Pursuit of a moving target

In the practice of constructing game scenarios, it is not uncommon for a situation where a persecuted object is constantly in motion. Since finding a path is not an instantaneous operation, frequent recalculation of the path with the Pursuer's stop looks like a persistent jerking of the pursuer. We in our arrangement have introduced the possibility of correcting the path without stopping the pursuer.

Let's imagine the pursuer is already on the move and follows the path to the target location. But during the movement of the pursuer, the target moved a considerable distance from the place where the path was found. In this case, you should call the RefinePath() function, giving as the argument a new target location. This function has one input parameter (similar to the MoveTo() function of type Vector3, or Transform).

An example of this feature may be found in the first demo scene (PathFinder3D/Scene1/Scripts/MissleController.cs). This function is used to correct the way of homing missiles.

Typically, a call to the path correction might look like this:

```
private void Update()
{
        //if the target has moved from the previous coordinate(targetOldPos) to more than
"targetPathUpdateOffset", update the path to the target
        if (Vector3.Distance(targetOldPos, target.position) > targetPathUpdateOffset)
        {
            targetOldPos = target.position;
            if (thisPursuerInstance.GetCurCondition() == "Movement")
                thisPursuerInstance.RefinePath(target);
        }
}
```

Finding a way outside the "Pursuer"

It is quite permissible that you need to find a way for your own targets between two points in space (not using this path inside Pursuer). You can do this using the Pursuer class, by calling FindWay().

The full definition of the function looks as follows:

```
void FindWay(Vector3 startPos, Vector3 targetPos, int pathfindingLvl, List<Vector3>
foundPath, PathfindingAlgorithm usingAlg, Action failurePathfindingActions = null,
Action succesPathfindingActions = null, bool inThreadOptimization = false, bool
inThreadSmoothing = false)
```

Arguments have the following meaning:

- Vector3 startPos - starting point coordinate
- Vector3 targetPos - the coordinate of the point to which you want to find the path
- int pathfindingLvl - the level of the spatial graph on which the pathfinding will be performed
- List<Vector3> foundPath - an instance of List <Vector3>, where the points of the found path will be written
- PathfindingAlgorithm usingAlg – the algorithm by which the pathfinding is performed can take the following values: PathfindingAlgorithm.AStar, PathfindingAlgorithm.waveTrace
- Action failurePathfindingActions - a delegate containing instructions for the case if it was not possible to find the path between two given points
- Action succesPathfindingActions – a delegate containing instructions for the case of a successful path finding
- bool inThreadOptimization - do we need to optimize the found trajectory?
- bool inThreadSmoothing - do we need to smooth the found trajectory?

The parameters failurePathfindingActions and succesPathfindingActions are needed in order for us to know whether or not the path was found. Since the search for a path occurs in another thread, the completion of the search function will be invisible to us if we do not use these parameters.

Examples of using:

Let’s imagine you have a game object that contains a customized Pursuer script, as well as another script that Pursuer will use solely to pathfinding.

```
using System;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(Pursuer))]
public class PathfindingScript : MonoBehaviour
{

    Pursuer thisPursuerInstance;
    List<Vector3> foundPath;
    private void Start()
    {
        thisPursuerInstance = gameObject.GetComponent<Pursuer>();
        foundPath = new List<Vector3>();
    }
    public void FindWay(Vector3 from, Vector3 to)
```

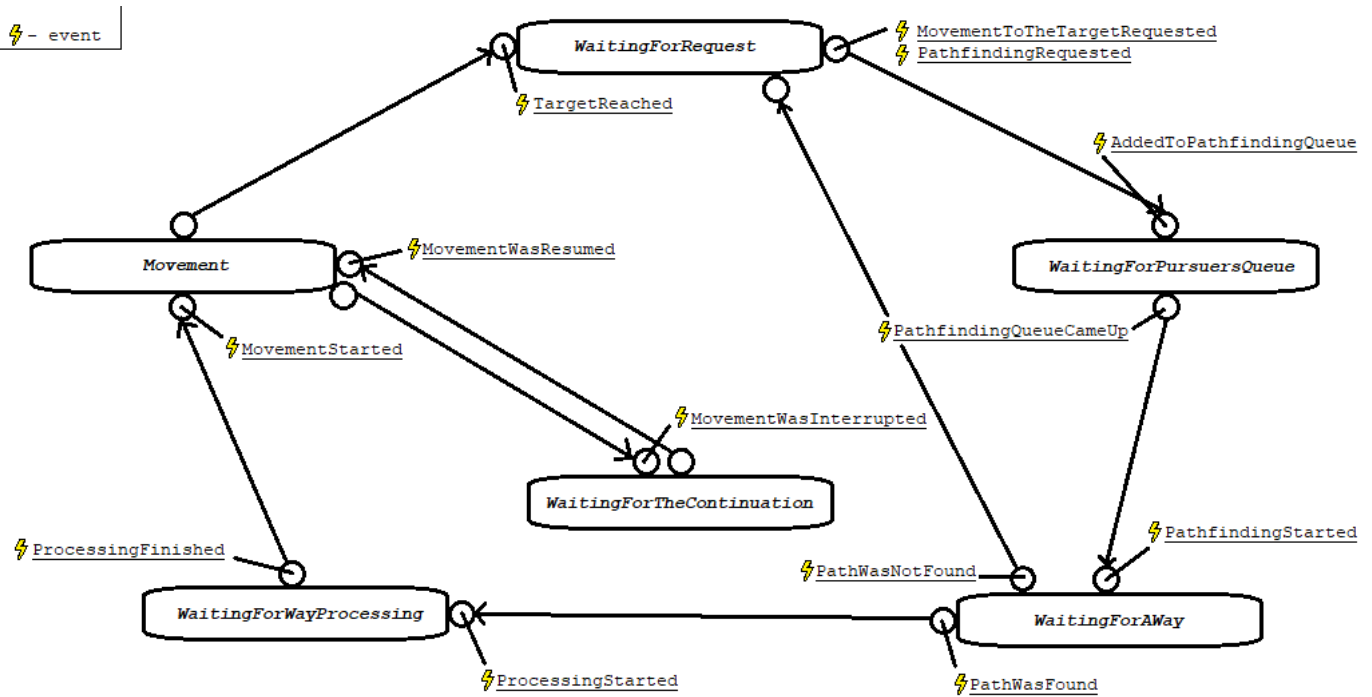
```
    {
        Action toDoAfterWayFound = new Action(() => {
gameObject.SendMessage("PursuerHasFoundAPath"); });
        Action toDoIfWayNotFound = new Action(() => {
gameObject.SendMessage("PursuerHasNotFoundAPath"); });
        thisPursuerInstance.FindWay(from, to, 0, foundPath,
PathfindingAlgorithm.AStar , toDoIfWayNotFound, toDoAfterWayFound, true, true);
    }
    public void PursuerHasFoundAPath()
    {
        Debug.Log("The way was found! The path contains the number of points = " +
foundPath.Count);
    }
    public void PursuerHasNotFoundAPath()
    {
        Debug.Log("There is no way between points");
    }
}
```

In fact, the case where there is no path between two points is rather difficult to detect, because the heuristic algorithm will try to find the path until the entire search space is exhausted. For this reason, the search can take a very long time. In order to avoid such a situation, you can manually stop the search path thread (for example, by timer). You can do this by calling the StopAllThreadingTasks () procedure.

Pursuer Class states and events

To simplify management and interaction with the Pursuer class, it includes a state machine and a system of events implemented using messages sent to the game object containing Pursuer among its components. More information about the mechanism of messages you can read in the corresponding section of the documentation - https://docs.unity3d.com/ru/current/ScriptReference/GameObject.SendMessage.html. Those, at the key moments in the Pursuer class, a corresponding message is sent to the game object on which the given Pursuer instance (gameObject.SendMessage ("messageText")) is running. For a list of all messages, see "Pursuer. User calls and events. "

Below you can see a state machine that reflects the normal mode of operation of the Pursuer class after calling MoveTo(). Transitions between states are accompanied by the triggering of certain events. The purpose of all events will be described in the table below.



The machine is implemented using the delegate:

```
delegate void condition();

volatile condition curCond;
```

,which takes as value one of the following procedures, refers to the state of the machine:

- void WaitingForRequest()
- void WaitingForPursuersQueue()
- void WaitingForAWay()
- void WaitingForWayProcessing()
- void WaitingForTheContinuation()
- void Movement()

This delegate is executed every time the FixedUpdate() event occurs. Each time by running delegate, a game object of the Pursuer receives one of the following messages:

- CondWaitingForRequest
- CondWaitingForPursuersQueue
- CondWaitingForAWay
- CondWaitingForWayProcessing
- CondWaitingForTheContinuation
- CondMovement

If the Pursuer is at rest and is not doing anything, then this equals to the state of "WaitingForRequest". This is the initial state of the Pursuer after the start of the scene.

After calling MoveTo(), the Pursuer goes to the "WaitingForPursuersQueue" state. It will remain in this state until the SpaceManager gives it permission to start searching for a path.

After receiving permission, the Pursuer will start searching for the path and go to the "WaitingForAWay" state. In this state it will remain until the path is found.

After finding the path, the Pursuer will process the pathfinding, i. e. will go to the "WaitingForWayProcessing" state. In this state, it will remain until the path is processed.

After the processing of the found path is completed, the Pursuer can start the movement and go to the "Movement" state. Being in this state, the Pursuer will follow the found path and exit this state only after reaching the end point of the path. Exiting this state means returning to the original state ("WaitingForRequest").

The Pursuer's movement can be suspended at any time by calling the InterruptMovement () method. Calling this method will move the Pursuer to the "WaitingForTheContinuation" state. You can resume movement by calling the ResumeMovement() method. This call will return the Pursuer to the "Movement" state.

From any state, the transition to the original state (WaitingForRequest) is possible. This is possible by calling the ResetCondition() method. Calling this method will also stop all active Pursuer threads.

SpaceManager. User calls and events.

As mentioned before, the SpaceManager should be present on the stage in a single instance. You can access this instance from any MonoBehaviour script. This is done most easily using the following call: Component.FindObjectOfType<SpaceManager>();

SpaceManager provides the following list of methods available to call from outside:

Method	Description	Arguments	Returning value
public bool PrimaryProcessing()	If ProcessAtStartup has a true value, the method will be called when the scene starts. Otherwise, it must be called manually. The completion of the primary processing is marked by sending the message "PrimaryProcessingFinished" to the game object on which SpaceManager is running, as well as sending the "TheGraphIsReady" message to all game objects on which there is a Pursuer script.	-	bool - will return true value if the initial processing has been started. Return false value if the primary processing has already been performed.
public float GetProcessingProgress()	Reasonable to call during the initial processing. Returns the current progress of the primary scene processing.	-	float - reflects the current progress of the primary processing. The range of values is [0.0f, 1.0f].
public int GetTotalTrisCountToProcess()	Reasonable to call during the initial processing. Returns the total number of triangles of all obstacles that will be processed.		int - the total number of triangles of all obstacles.
public int GetCurentProcessedTrisCount()	Reasonable to call during the initial processing. Returns the current number of already processed triangles.	-	int - the current number of triangles already processed.
public void ProcessAnObstacle(GameObject obstToProc)	It recalculates the cells occupied by the obstacle. Cells occupied by an obstruction earlier are released from it. It makes sense to call for obstacles that appear after the initial processing. Either for obstacles that change position, size or turn during the game.	GameObject obstToProc - Game object obstacles that need to be processed.	-
public void RemoveAnObstacle(GameObject obstToRem)	Produces the release of cells occupied by an obstacle. This method can be useful in case the obstacle is destroyed or become invalid.	GameObject obstToRem - game object obstacles, cells occupied by which you need to	-

		release.	

Pursuer. User calls and events.

For your convenience, we have added into the "Pursuer" Class a number of procedures and functions that make it easier to control the pursuer and the mechanism of interclass interaction.

Here is a table that fully discloses the purpose and functionality of these procedures and functions.

public float GetTotalPathLength()	Description	Returns the length of the last found path.
	Arguments	-
	Returning value	float - path length taking into account optimization and smoothing in co-ordinates unity.
public List<Vector3> GetFoundPath()	Description	Returns the list of vertices of the last found path without taking into account optimization and smoothing.
	Returning value	List<Vector3> - list of trajectory vertices.
public List<Vector3> GetFinalPath()	Description	Returns the list of vertices of the last found path, taking into account optimization and smoothing (if these options are enabled).
	Returning value	List<Vector3> - list of trajectory vertices.
public bool SetConstraints(float xMin, float xMax, float yMin, float yMax, float zMin, float zMax)	Description	Sets the position and size of the area in which the path can be searched.
	Arguments	float xMin, float xMax - the width of the region along the x axis, and so on.
	Returning value	bool - will return a false value if at least one pair of values has error values (Min> = Max). Returns true if new values are successfully set.
public string GetCurCondition()	Description / Returning value	Returns the name of the method of the currently assigned state to the delegate. (see section "Pursuer class states and events")
float GetCurWayProgress()	Description / Returning value	Returns the ratio of the lengths of the already passed part of the path to the entire path.
public bool RefinePath(Transform /	Description	This function should be used to correct the path to the target in the event

Vector3 target)		that the target has changed its location during the pursuit. After the call, the pursuer begins recounting the part of the path that has not been passed yet, continuing to move. After the conversion is completed, the pursuer will be rebuilt to a new trajectory.
	Arguments	Transform / Vector3 target - a new target position.
	Returning value	bool - will return a false value if the pursuer is not in the "Movement" state, i.e. does not pursue any target. Returns true if the path correction has started successfully.
public void MoveTo(Transform / Vector3 target, bool topPriority = false)	Description	This method should be used in case you want to start moving to the target. (see the section "States and events of the Pursuer class"). When you call this method, the following checks will be performed: • Is the target within the search area • Is the pursuer inside the search area • Is the target in the cage occupied by an obstacle • Is the pursuer in a cage occupied by an obstacle A negative result of at least one check will cancel the call of this method and result in sending the corresponding message to the pursuer’s game object (see below)
	Arguments	1. Transform / Vector3 target - the target (coordinate) to which you need to find the path and start the movement. 2. bool topPriority = false - This parameter specifies the order of the start of the search path. True value will lead to the search of the path to bypass the queue of pursuers. A false value will lead to the putting of the pursuer in the path search queue (regular script). The search will begin when the pursuer reaches his turn. (see section "Pursuer class states and events")
public bool InterruptMovement()	Description	Suspend the pursuer's movement, by transferring to the "WaitingForTheContinuation" state. (see section "Pursuer class states and events")

	Returning value	bool - will return a false value if the pursuer is not in the "Movement" state. If the motion is suspended successfully, the true value will return.
public void ResetCondition()	Description	You should call to transfer the persecutor to the original state ("WaitingForRequest"). Stops all active threads, cleans all local variables, removes the pursuer from the queue (if in queue).
public bool ResumeMovement()	Description	Will resume the movement of the pursuer, by transferring it to the "Movement" state. (see section "Pursuer class states and events")
	Returning value	bool - will return a false value if the pursuer is not in the "WaitingForTheContinuation" state. In case of successful resumption of movement, the true value will return.
public void StopAllThreadingTasks()	Description	Stops all active threads, cleans all local variables, removes the pursuer from the queue (if in queue). The application of this procedure is described in section "Finding a path outside of Pursuer."
public void FindWay(Vector3 startPos, Vector3 targetPos, int pathfindingLvl, List<Vector3> foundPath, PathfindingAlgorithm usingAlg, Action failurePathfindingActions = null, Action succesPathfindingActions = null, bool inThreadOptimization = false, bool inThreadSmoothing = false)	For details on how to use this method, see "**Finding a path outside of Pursuer.**"	

Also, an event system based on the mechanism for sending messages to a game object (an object containing this instance of the Pursuer class) is introduced into the Pursuer class. More information about the mechanism of messages you can read in the corresponding section of the documentation - https://docs.unity3d.com/ru/current/ScriptReference/GameObject.SendMessage.html.

All messages are presented in the table below with explanations.

Message text	Conditions of the message

EventAddedToPathfindingQueue	Occurs after the pursuer stood in line for a search for the path.
EventPathfindingQueueCameUp	Occurs after the pursuer in the order of the queue has received permission to find a way.
EventPathfindingRequested	Occurs when the MoveTo () function is called after successfully passing all the checks for the possibility of finding a path between two points.
EventPathfindingFromOutwardPointRequested	Occurs after a call to MoveTo () if the pursuer is outside the search area specified for it.
EventPathfindingToOutwardPointRequested	Occurs after a call to MoveTo () if the target is outside the search area specified for the pursuer.
EventPathfindingFromStaticOccupiedCellRequested	Occurs after a call to MoveTo () if the pursuer is in a cage occupied by some obstacle.
EventPathfindingToStaticOccupiedCellRequested	Occurs after a call to MoveTo () if the target is in a cage occupied by some obstacle.
EventPathfindingStarted	Occurs when the thread of path searching starts and the pursuer's state is "WaitingForAWay".
EventProcessingStarted	Occurs after the path was found, at the time the path processing thread was started.
EventProcessingFinished	Occurs when the path processing thread ends.
EventMovementToTheTargetRequested	Occurs when the MoveTo() is called.
EventMovementStarted	Occurs when the processing thread completes and the pursuer moves to the "Movement" state.
EventMovementWasInterrupted	Occurs after a successful call to InterruptMovement() when the pursuer moves to the "WaitingForTheContinuation" state.
EventMovementWasResumed	Occurs after a successful ResumeMovement () call when the pursuer changes to the "Movement" state.
EventTargetReached	Occurs when the pursuer reaches the last point of the path by moving the pursuer to the "WaitingForRequest"

	state.
EventPathWasFound	Occurs after the completion of the path searching when the pursuer moves to the "WaitingForWayProcessing" state. Signifies the successful finding of a path between two points.
EventPathWasNotFound	Occurs after the completion of the path searching, when the pursuer moves to the "WaitingForRequest" state. Signifies the fact that the path between two points could not be found.
EventPathRefiningRequested	Occurs after a call to RefinePath ().
EventPathRefiningStarted	Occurs after a call to RefinePath () when the thread of path searching starts.
EventPathWasRefined	Occurs after the successful finding of the correcting section of the trajectory and the recording of the updated path.
EventPathWasNotRefined	Occurs if the correction section of the trajectory could not be found.

For any questions, suggestions, comments, you can contact us by e-mail:

support@gracefulalgs.com

The Asset page on Unity Asset Store:

https://assetstore.unity.com/packages/tools/ai/pathfinder-3d-100285

Here is the corresponding forum thread:

https://forum.unity.com/threads/pathfinder-3d-pathfinding-in-three-dimensional-space.499144/

The Asset page on our Website:

https://gracefulalgs.com/portfolio/pathfinder/

Always happy to help you :)