

User Guide

AStar 2D

A complete pathfinding solution for 2D grid based games

Trivial Interactive

Version 1.1.7

AStar 2D is a complete pathfinding solution for 2D tile or grid based Unity games. It is ideally suited to top down tile based games and tower defence games but can easily be adapted to work in other scenarios.

This user guide is for the updated version of AStar 2D (1.1.0 and onwards). Support for previous versions of AStar 2D will be dropped in favour of this latest version.

Features

- Quick and easy setup / integration into existing projects.
- Includes easily expandable agent class for following paths.
- Dynamic route updates for changing environments.
- Supports any number of search grids.
- Assign different grids to different agents to shrink the search space.
- Enable / Disable diagonal movements in the click of a mouse.
- Allows diagonal corner cutting to be prevented.
- Offload the expensive pathfinding calculations to separate threads to maintain maximum performance (All done behind the scenes, works out of the box).
- Includes visualisation tools to help with debugging.
- Includes example scenes and stress tests.
- All scripts use custom namespaces to prevent clashing type names.
- Comprehensive .chm documentation of the API for quick and easy reference.
- Fully commented C# source code included.

Quick Start

If you are just interested in getting the asset setup and integrated into your project as quickly as possible then skip ahead to the Integration section that provides a step by step guide of the process.

As an alternative, you can expand on one or more of the demo scenes provided using the attached scripts and prefabs as a foundation. This method requires almost no knowledge at all of scripting as the solution works out of the box and is easily modifiable to suit many different applications.

Examples

Included in the package is a number of demo scenes that show the features of AStar 2D. The demo will typically contain an array of pathfinding nodes represented as blue sprites which represents the game tile map. Each node can be clicked with the right mouse button to toggle its walkable status.

Usage

- Load a demo scene in the Assets/Demo folder and start the player
- You will see an array of nodes distributed across the screen.
- Use the right mouse button to change the walkable status of a node. (Create obstacles).
- Instructions specific to the demo are shown in the top left hand corner of the screen.
- You can open the AStar 2D performance visualizer window to view real-time information about the performance of the algorithm. (Window/AStar 2D Performance Visualizer)

Further Settings

When the scene is running in game mode there are a few settings that can be modified to help visualise the node network. Select the 'TileManager' object from the scene hierarchy to edit these values either in the scene view or at runtime.

- Diagonal Mode – The diagonal method used when finding paths. 'No Diagonal' means that diagonal paths will not be considered. 'Diagonal' means that diagonal paths will be considered. 'Diagonal No Cutting' means that diagonal paths will be considered however the path will not cut across non-walkable nodes when changing direction.
- Allow Threading – Allows the bulk of the pathfinding work to be offloaded to a separate thread to prevent the main thread from being blocked by expensive calculations
- Visualize grid – Creates a dedicated render component to display the node connections in the game view. See the visualization tools section for more information.
- Visualize path – Creates a dedicated render component to display the last path found by the grid. See the visualization tools section for more information.

Namespaces

All of the classes and structures within the AStar 2D asset are organised into their own appropriate namespaces to avoid name clashes when integrated to projects.

- **AStar_2D:** This is the core namespace and must be imported to access the main classes that are exposed to the user. This is the only namespace you will need to import to access pathfinding within Unity. It also contains useful classes for interfacing with the system such as a grid index class and a mono delegate class.
- **AStar_2D.Collections:** This namespace only contains internal data structures that the user need not use, however if the user can make use of them within their own project then they are more than welcome to utilize the classes within.
- **AStar_2D.Demo:** This namespace contains all of the scripts that are used by the demo scenes and show how to use the different features that AStar 2D offers.
- **AStar_2D.Pathfinding:** This is the namespace that contains classes and structures to implement the low level algorithm. It also contains the 'SearchGrid' class which provides standalone pathfinding functionality without the need to Unity components.
- **AStar_2D.Threading:** This namespace is where all the threading related classes and structures are stored and need not be accessed by the user.
- **AStar_2D.Visualisation:** This namespace contains all the editor aids that allow the node grid to be represented visually. These classes are helper classes and should not be used directly by the user. Instead all of these aids can be accessed through the Unity Editor.

Using Call-backs

Since AStar 2D makes use of a number of worker threads to perform the pathfinding tasks, it is not possible to directly call a method to return a path unless you use the immediate method which avoids threading. Instead the user must provide a call-back to the search which will be invoked once the thread has finished working on the task. The delegate is guaranteed to always be called from the main thread so any you are able to interact with the Unity API in the call back methods. The call-backs used throughout the API take on two forms:

C# Delegate

The default method makes use of built in C# delegates to call the user supplied method. The only requirement is that the user method conforms to the 'PathCompleted' delegate, specifically making sure that the arguments match. C# delegates are recommended over Mono Delegates as they offer much better performance however Mono Delegates can be more suited to the Unity workflow.

Mono Delegate

The API also provides an alternative call-back method that makes use of built in Unity functionality. This is known as a Mono Delegate and allows the user to pass a specific Mono Behaviour script along with the method name as a string. Internally it then makes use of the 'SendMessage' functionality of Unity to trigger the desired method.

For further information on call-backs, it is recommended to look at the example scripts and how they work, or alternatively take a look at the include API documentation for examples.

Concepts

AStar Grid

An AStar Grid is a component that can be attached to game objects and represents a two dimensional array of nodes that is able to perform pathfinding tasks. There may be any number of AStar Grids in a single scene and you are able to request paths directly from individual grids. Alternatively you can make use of the agent class in which case an AStar Grid can be assigned directly to the agent. Each of these individual nodes will have specific properties related to pathfinding such as whether they are walkable and their layout is defined by their index into the array. These components will typically receive all pathfinding requests and generate paths that conform to the settings specified in the Unity Editor.

Path

AStar 2D uses the idea of 'Paths' which are essentially a sequence of nodes that when followed in order will lead to a specific destination. Whenever you want to issue a pathfinding request to AStar 2D you will get a Path as a result and as such, there are some helpful attributes to a path that aim to make it as simple as possible to reach the destination.

Paths are able to determine whether the destination is still reachable after the initial pathfinding request has completed. This allows dynamic routing behaviour if an existing path becomes blocked as well as dynamic obstacle avoidance. You can take a look at the Agent script to see how an agent is able to detect and avoid newly created obstacles using dynamic routing.

Paths also provide functionality to check whether an objects transform component has reached a specified node in the path. This can be useful to determine when the agent has reached the target node and should no longer attempt to move towards its target.

Agent

An agent is an autonomous entity which is able to request and follow paths without input. The agent simply requires a target destination and all of the complexities of navigating and moving towards that destination are handled automatically. AStar 2D provides a few example agents which can be found in the demo folder, each with slightly differing abilities. Although agents are a strong presence in the AStar 2D package, they are not a requirement and you are able to implement your own behaviour if needed.

AStar 2D uses agents in order to provide the base Ai behaviour associated with pathfinding such as requesting paths based upon its current location, and following paths to reach a target destination. AStar 2D provides all of the basic functionality in a way that can be easily expanded upon through inheritance or composition. Each agent may be assigned an AStar Grid which will be used for all pathfinding requests, or alternatively the agent will select an appropriate grid when the game starts (Typically the first grid added to the scene).

For more information about AStar 2D agents, take a look at the demo assets and scenes to see how they work. There are also a number of demo scripts that aim to show how the Agent class can be expanded upon to add things such as animation or roaming behaviour.

Integration

The following steps will allow you to setup AStar 2D to work with your existing project. You can also take a look at the demo scripts included with AStar 2D to see how the example pathfinding works.

1. Import the package into unity.
2. Implement the 'IPathNode' interface. Tile based games will typically have a specific class that represents a single tile and its properties. This class would be where the IPathNode implementation would be added. Since the 'IPathNode' requirement is an interface, the tile class can still inherit from mono behaviour as usual. See figure 1 for an example implementation. The interface members are described below:

IsWalkable: Can the tile be walked on. Solid tiles such as walls or obstacles should return false and they will be excluded from the final path.

Weighting: A value between 0 and 1 that represents the extra cost of including this tile in the path. If you want a tile to be walkable but have a high resistance then you can use this value. An example could be a water tile which may be passable but the algorithm should try to find a better path before including water tiles. This alternative path may be longer but will ultimately have a lower cost. If you do not require weightings in your game then simply return a value of 0.

WorldPosition: If you represent each tile as a game object then you can simply return the 'transform.position' for this property. Otherwise you will need to calculate the position when the tile is created. The position is used by agents to move between path node locations.

```
1 using UnityEngine;
2 using System.Collections;
3
4 using AStar_2D;
5
6 public class ExampleTile : MonoBehaviour, IPathNode
7 {
8     public bool IsWalkable
9     {
10         get { return true; }
11     }
12
13     public float Weighting
14     {
15         get { return 0; }
16     }
17
18     public Vector3 WorldPosition
19     {
20         get { return transform.position; }
21     }
22 }
23
```

Figure 1

Note: Make sure you add the using statement to the script to access the AStar 2D classes.

3. Create an array of tiles using the class from the previous step. If you are adding AStar 2D to an existing game then it is likely that you will already have an array of tiles. The must be 2 dimensional and contain a valid non-null instance of the tile class for each element. The array must also have a width and height that is greater than 0. Figure 2 shows an example tilemap implementation using the 'ExampleTile' class from the previous step:

```
1 using UnityEngine;
2 using System.Collections;
3
4 using AStar_2D;
5
6 public class ExampleTilemap : MonoBehaviour
7 {
8     public GameObject tilePrefab;
9     private ExampleTile[,] tiles = new ExampleTile[8, 8];
10
11     public void Start()
12     {
13         // Create each tile
14         for(int x = 0; x < 8; x++)
15         {
16             for(int y = 0; y < 8; y++)
17             {
18                 // Create an instance of the tile prefab
19                 GameObject go = Instantiate(tilePrefab, new Vector3(x, y), Quaternion.identity) as GameObject;
20
21                 // Get the tile component
22                 tiles[x, y] = go.GetComponent<ExampleTile>();
23             }
24         }
25     }
26 }
27
```

Figure 2

4. Pass the tile array to AStar 2D. This is an important step and must be done for every pathfinding grid. There are 2 ways in which you can achieve this which are inheritance, or by storing a reference.

Inheritance

This is the recommended method and requires that you inherit from the 'AStarGrid' class. Typically this will be done by the class that contains the tile array, in our case the 'ExampleTilemap' class from step 3. The 'AStarGrid' class is a component and inherits from 'MonoBehaviour' so in many cases the class can be used as a direct replacement for the 'MonoBehaviour' class. Figure 3 shows the amended 'ExampleTilemap' class using inheritance.

```
1 using UnityEngine;
2 using System.Collections;
3
4 using AStar_2D;
5
6 public class ExampleTilemap : AStarGrid
7 {
8     public GameObject tilePrefab;
9     private ExampleTile[,] tiles = new ExampleTile[8, 8];
10
11     public void Start()
12     {
13         // Create each tile
14         for(int x = 0; x < 8; x++)
15         {
16             for(int y = 0; y < 8; y++)
17             {
18                 // Create an instance of the tile prefab
19                 GameObject go = Instantiate(tilePrefab, new Vector3(x, y), Quaternion.identity) as GameObject;
20
21                 // Get the tile component
22                 tiles[x, y] = go.GetComponent<ExampleTile>();
23             }
24         }
25     }
26 }
27
```

Figure 3

Once you have inherited from the 'AStarGrid' class then you will be able to access its methods. You will also be able to configure the pathfinding settings from the Unity Editor as seen in figure 4. The method that initializes the pathfinding grid is called 'constructGrid' and takes an array argument which will be the array of tiles you created in step 3. Figure 5 shows the 'ExampleTilemap' class which has been amended to call this method during the 'Start' method.

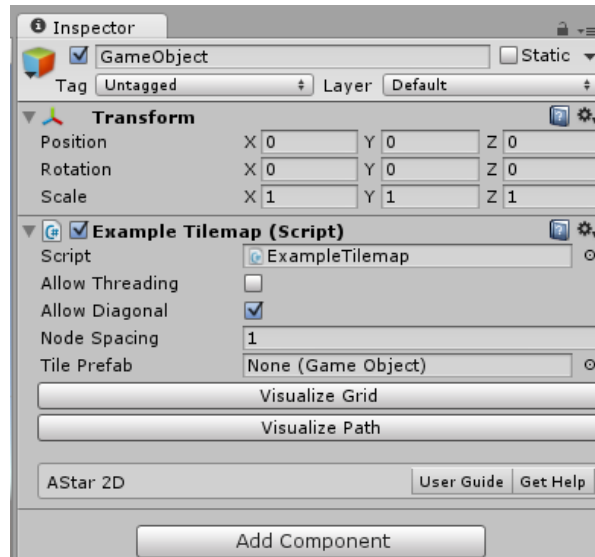


Figure 4

```

1  using UnityEngine;
2  using System.Collections;
3
4  using AStar_2D;
5
6  public class ExampleTilemap : AStarGrid
7  {
8      public GameObject tilePrefab;
9      private ExampleTile[,] tiles = new ExampleTile[8, 8];
10
11     public void Start()
12     {
13         // Create each tile
14         for(int x = 0; x < 8; x++)
15         {
16             for(int y = 0; y < 8; y++)
17             {
18                 // Create an instance of the tile prefab
19                 GameObject go = Instantiate(tilePrefab, new Vector3(x, y), Quaternion.identity) as GameObject;
20
21                 // Get the tile component
22                 tiles[x, y] = go.GetComponent<ExampleTile>();
23             }
24         }
25
26         // Pass the tiles to AStar 2D
27         constructGrid(tiles);
28     }
29 }

```

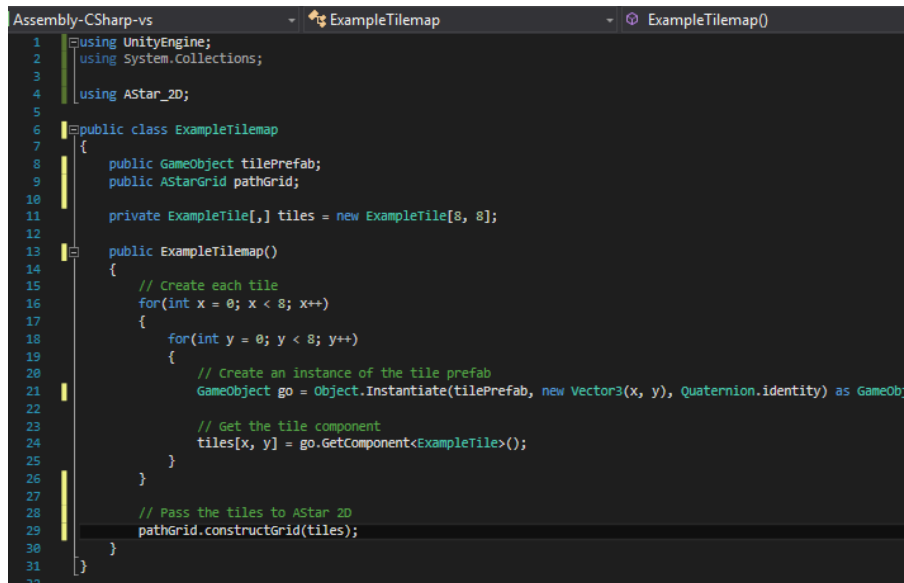
Figure 5

Reference

This second method is not as flexible as the inheritance method but means that you are able to store the tile array in another class that does not need to be a Unity component.

Instead of inheriting from the 'AStarGrid' class, we simply need to store a reference to it which will typically be assigned in the Unity editor to a public variable. The 'AStarGrid' class inherits from 'MonoBehaviour' so it does need to be added to the

scene. Figure 6 shows the amended 'ExampleTilemap' class which now references an 'AStarGrid' component.



```
1 using UnityEngine;
2 using System.Collections;
3
4 using AStar_2D;
5
6 public class ExampleTilemap
7 {
8     public GameObject tilePrefab;
9     public AStarGrid pathGrid;
10
11     private ExampleTile[,] tiles = new ExampleTile[8, 8];
12
13     public ExampleTilemap()
14     {
15         // Create each tile
16         for(int x = 0; x < 8; x++)
17         {
18             for(int y = 0; y < 8; y++)
19             {
20                 // Create an instance of the tile prefab
21                 GameObject go = Object.Instantiate(tilePrefab, new Vector3(x, y), Quaternion.identity) as GameObject;
22
23                 // Get the tile component
24                 tiles[x, y] = go.GetComponent<ExampleTile>();
25             }
26         }
27
28         // Pass the tiles to AStar 2D
29         pathGrid.constructGrid(tiles);
30     }
31 }
32
```

Figure 6

Note: The creation of the tile map is now done from the constructor since the class no longer inherits from MonoBehaviour. We also pass the tiles to AStar 2D using the stored AStarGrid reference.

5. Create a tile prefab. If you are adding AStar 2D to an existing game then you will probably have already completed this step.

The 'ExampleTilemap' class references a game object that is used for instantiating tiles at startup. You simply need to create a prefab that has the script from step 2 attached. It may also be worth creating a visual representation of the tile using a sprite or mesh but is not essential. Once you have created the prefab then you can assign it to the 'tilePrefab' variable of the 'ExampleTilemap' component within the Unity Editor.

6. That's it! You are all setup and pathfinding has now been added to your game. You can drop an Agent prefab into the scene which are located in 'AStar 2D/Demo/Prefabs' and you can simply call the method 'setDestination' on the Agent script which will find a path to the destination.

If you drop the 'RoamingAgent' prefab into the scene then you should see that the agent will automatically select a random target destination and proceed to move to that location.

Next steps

Once you have successfully setup AStar 2D you can immediately start using the features of the API. If you are not sure on the different arguments to supply to the methods or how to handle the paths once received it is recommended to take a look at the complete API documentation included within the asset. This will allow you to find out how to use a specific method and what it actually does. The documentation is included inside the asset package when downloaded from the asset store.

Receiving errors after setup?

If you are receiving error messages after initial setup and don't know why, it could be something simple to fix. One of the most common setup errors is shown in figure 7 and occurs when you attempt to request a path before you have constructed the pathfinding grid.

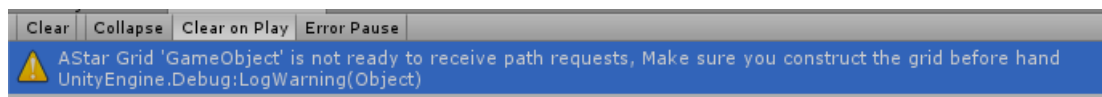


Figure 7

This issue can be fixed simply by ensuring that the grid has been initialized before issuing any pathfinding requests. You can check if a grid has been setup by calling 'IsReady' on an 'AStarGrid' instance. In our example, the construction of the grid is performed in the Unity 'Start' method which may cause issues since the 'RoamingAgent' script also requests a path in the 'Start' method. Depending upon the order of execution, you may receive the previous error. If so then you can either change the script execution order in the Unity settings or simply move the grid construction into the 'Awake' method.

Visualisation Tools

Included in the Astar 2D asset are a number of useful tools aimed to make debugging and performance monitoring easier. These tools are only available within the unity editor and the analysing code will be stripped out for standalone builds to achieve the best performance.

Trouble viewing lines?

All lines in AStar 2D are rendered using the LineRenderer component where possible which can sometimes appear behind meshes or sprites depending on your scene setup. If you are having trouble viewing the lines you can simply find the line renderer game object responsible for rendering the lines and modify the Z position so that lines are rendered in front of the scene.

Path View

The path visualizer is a script component that renders the last path generated from the search. It does this by drawing a line between each node using the path created by the search providing a visual representation of the path.

You can enable path visualisation by first selecting the 'AStarGrid' object in the hierarchy and then clicking 'Visualize Path' in the inspector window (Figure 8). You will notice that a game object is created in the scene with the same name as the 'AStarGrid' object with a visual identifier added to the end and the path will be rendered in the game view.

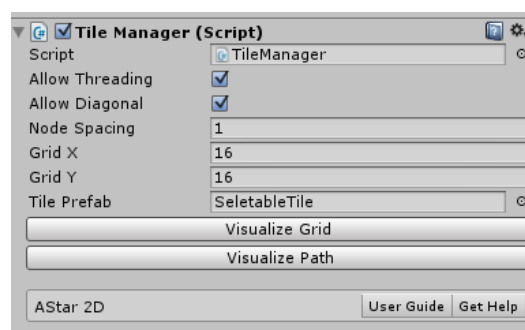


Figure 8

Figure 9 shows the example 'Demo' scene making use of path visualisation.

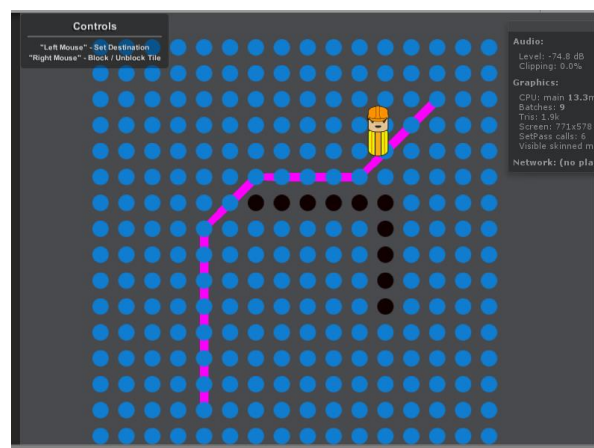


Figure 9

Grid View

The grid view is a script component that is able to provide a visual representation of how each node within the grid connects to its neighbours. Each line rendered by the component represents a valid path to a neighbouring node. This means that you can make sure that a nodes properties are updated correctly when expected such as modifying the walkable property at runtime.

You can enable grid visualisation using the same process as path visualization, by first selecting the 'AStarGrid' object in the hierarchy and then clicking 'Visualize Grid' in the inspector window. You will notice that a game object is created in the scene with the same name as the 'AStarGrid' object with a visual identifier added to the end and the grid will be rendered in the game view.

Figure 10 shows the example 'Demo' scene making use of grid visualization.

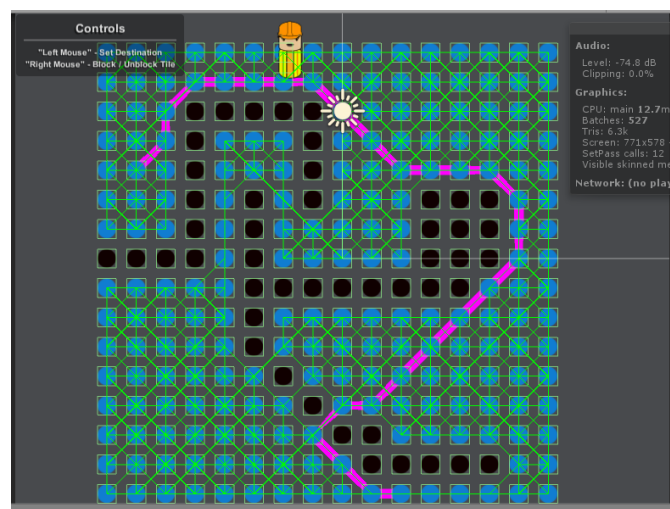


Figure 10

Trouble viewing lines?

Due to the way the grid view is laid out, the visual line aids cannot be rendered using a Line Renderer and are instead rendered using 'Debug.DrawLine'. This method requires that gizmos are enabled in order for the lines to appear in the game view.

Performance View

The performance view is an editor extension that is able to provide data regarding the performance of the algorithm and how the thread manager is distributing the requested tasks between the worker threads responsible for the requests. It is a useful aid to help determine the worst case performance scenario and to ensure that you do not overwork the threads which can result in delayed requests and backlog.

The Performance view can be accessed by the drop down window menu. Select AStar 2D Performance Visualizer to show the window

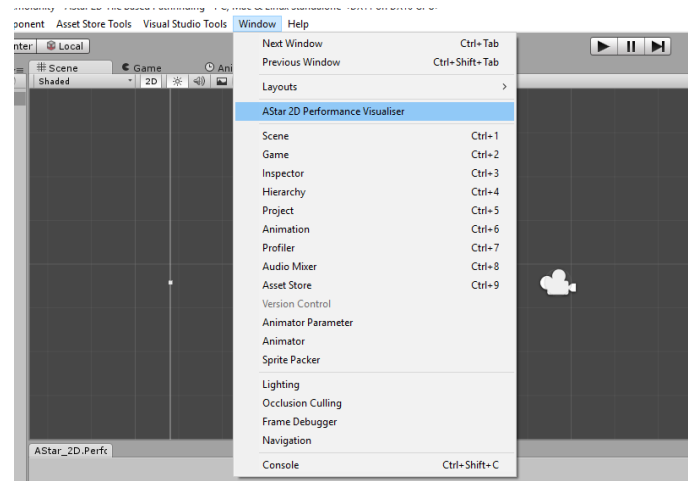


Figure 11

Once you open the performance visualizer you should see a window as shown in figure 12:

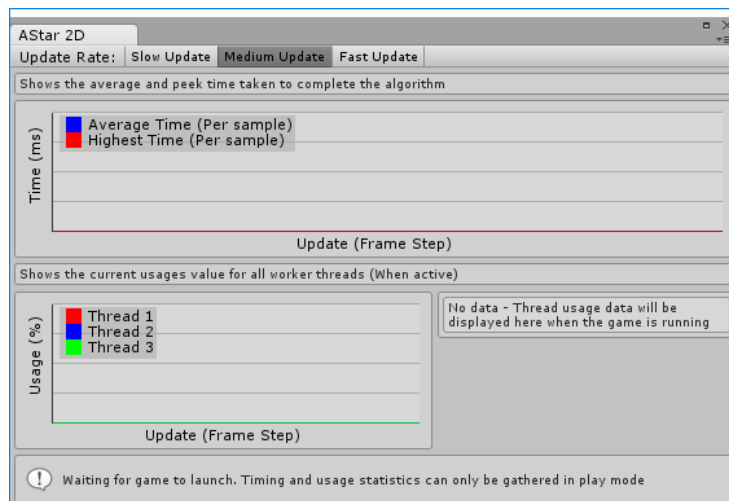


Figure 12

The window displays 2 charts that update in real-time based on the performance of the algorithm, along with thread usage statistics that provide information about the workload of individual threads. The window will not display any data until the game has started and pathfinding requests are issued. In order to see the functionality of the window, it is recommended that you open the 'StressTest'

scene included in the demo where you are able to spawn many agents that will automatically issue pathfinding requests as needed. Figure 13 shows data recorded from the 'StressTest' scene.

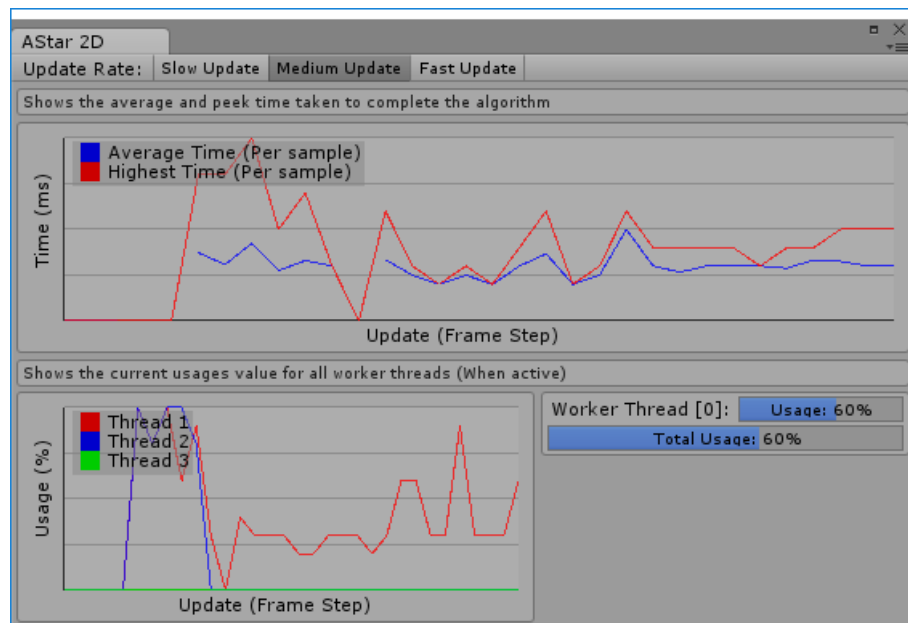


Figure 13

The following section will cover the data that is recorded by the visualizer and what it represents:

Chart 1

- **Average Time (Per Sample):** This value is recorded every time a thread completes a pathfinding request and represent the average amount of time in milliseconds that the algorithm took to complete. The chart range is from 0 to 12 milliseconds where anything over this amount is considered as overstress and alternative worker thread may be spawned to handle requests.
- **Highest Time (Per Sample):** This value is recorded every frame and represent the single request in the sample that took the longest time to complete. Peaks and spikes indicate a particularly long search time.

Chart 2

- **Thread 1:** Shows the average thread usage value as a percentage. When threading is enable, there will always be at least 1 active thread which will be thread 1. Requests that cause overstress to the thread may result in extra worker threads being spawned depending upon settings used.
- **Thread 2:** Shows the average thread usage for the second worker thread as a percentage. The value will default to zero when the thread is not active or has not been spawned.
- **Thread 3:** Shows the average thread usage for the third worked thread as a percentage. The value will default to zero when the thread is not active or has not been spawned.

Usage

The usage section shows the actual real-time usage of the threads that are currently active. The value may appear to flicker as the usage fades in and out quickly as the value is determined by the number of queued requests each thread must complete. It is useful to see an overview of the current performance of AStar 2D and how much more stress the system can handle. The total usage value represents the overall usage based on configuration settings.

There are scenarios where the demand for pathfinding tasks is too high for the worker threads and it is unable to complete the tasks quick enough. This is the worst case scenario because if the task queue continues to fill then the threads will be backlogged with requests and will eventually become totally out of sync with the main thread resulting in very large delays between tasks being accepted and completed. In simple terms, you are asking the thread to process too much information. This can easily be detected in the performance view by an exponential growth of the average time per task until the point of no return. If this is the case, it is safe to assume that you have explored the maximum performance potential of the system and should reduce the number of pathfinding requests where possible.

Figure 14 shows a typical case where pathfinding for 2000 agents results in total overstress.

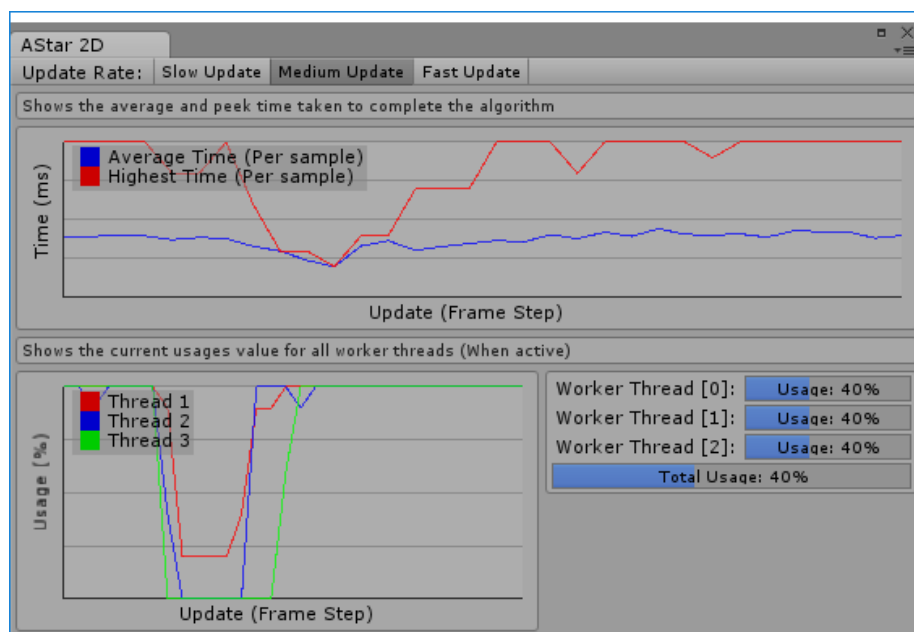


Figure 14

Note: The current usage display indicates that the threads are not overworked and can handle more load. This is because the number of requests waiting can be completed before the threads execution time is up, however this does not take into account requests that are issued while the thread is working which could be many. The average usage chart gives a much better representation of the actual usage.

The visualizer window also incorporates a toolbar where you are able to modify the update frequency of the sampler. By selecting slow, medium or fast update, the number of samples per frame will be modified to display either more or less information. The faster the samples update, the more accurate the data will be but it can have more of an impact of the performance of the game.

The performance visualizer window will only output information when the game is running. Until then, the window will display a message stating that there are no results available until the game is started.

It is also worth noting that all the code monitoring the performance of the system will be stripped out for builds, even if they are set as debug builds. This is because the code does lots of costly calculations to accurately determine the performance of the worker thread and as a result could directly impact the performance of your game.

Advanced Usage

Custom Node Connections

AStar 2D allows you to implement your own node checks when the algorithm is running. This allows you to dynamically modify the allowed connections between nodes based on the current state of the game. This can be very useful when you want to block access to a target node from another node, but do not want the target node to be un-walkable.

In order to add these checks into your game you will need to inherit from the SearchGrid class which is where the main AStar implementation is located. Once you have a derived class you will then be able to override a method named 'validateConnection' which will be called while the algorithm is running. You should take care as to not call any expensive methods from this method as it may be called many thousands of times per search, depending upon the complexities of the search grid.

The behaviour of this method is simple in principle. The return value is a Boolean which should be true when the connection between the nodes is allowed, and false when the connection is not allowed. Included in the demo is an example script named 'SelectiveSearchGrid' which provides a basic example of this method. The demo script simply checks the index of the incoming nodes to see whether the neighbouring node is at a lower Y index. This has the result in game of preventing the agent from moving down the grid. The agent will simply return a fail result indicating that the destination is not reachable.

Report a Bug

At Trivial Interactive we test our assets thoroughly to ensure that they are fit for purpose and ready for use in games but it is often inevitable that a bug may sneak into a release version and only expose its self under a strict set of conditions.

If you feel you have exposed a bug within the asset and want to get it fixed then please let us know and we will do our best to resolve it. We would ask that you describe the scenario in which the bug occurs along with instructions on how to reproduce the bug so that we have the best possible chance of resolving the issue and releasing a patch update.

<http://trivialinteractive.co.uk/bug-report/>

Request a feature

AStar 2D was designed as a complete pathfinding solution, however if you feel that it should contain a feature that is not currently incorporated then you can request to have it added into the next release. If there is enough demand for a specific feature then we will do our best to add it into a future version. Please note, this is a pathfinding asset and requested features should fall under this category. We will make no attempt to add features that are off topic such as enemy AI behaviours.

<http://trivialinteractive.co.uk/feature-request/>

Contact Us

Feel free to contact us if you are having trouble with the asset and need assistance. Contact can either be made by the contact options on the asset store or buy the link below.

Please attempt to describe the problem as best you can so we can fully understand the issue you are facing and help you come to a resolution. Help us to help you :-)

<http://trivialinteractive.co.uk/contact-us/>