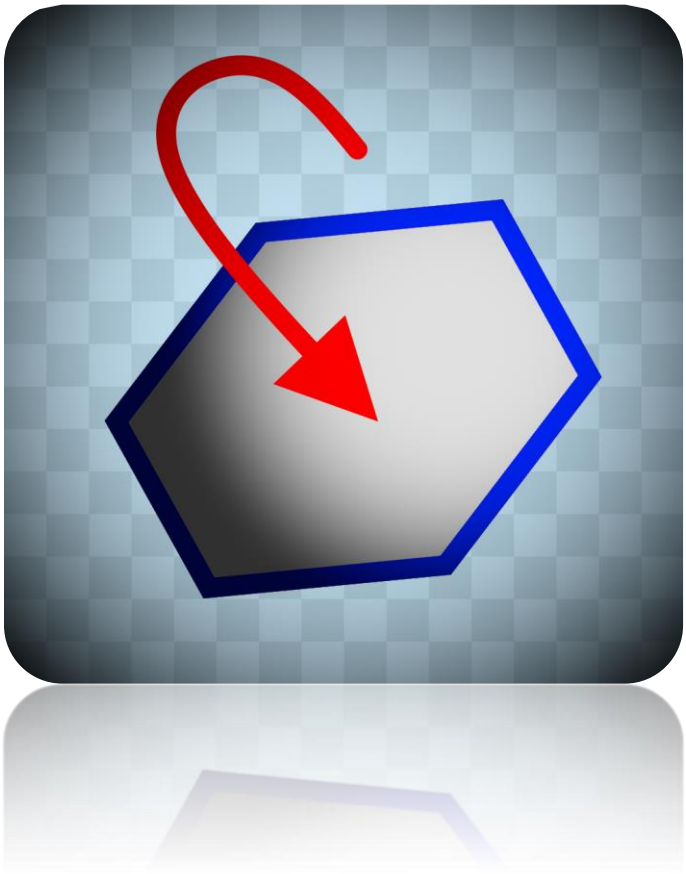
Simple Tile Pathfinding

Reference Document

By TednesdayGames
Contact: tyler@tymax.com



Intro

Simple Tile Pathfinding is a *Tile* based, pathfinding solution that garners useful results in just a few lines of code. It provides comprehensive functionality that intends to stay out of the way and allows the designer (you) to get on with the task at hand.

What is it?

Simple Tile Pathfinding works as a single component attached to a *Grid*, which uses a variety of additional classes that exist under a single *Simple Pathfinding* namespace. It utlizes a heuristic A* algorithm to perform shortest path searches between two points on a *Tilemap*. *Simple Tile Pathfinding* is to be used with a *Grid* and *Tilemap* for the best results.

There are number of script examples that use *Simple Tile Pathfinding.* These are located in the Demo folder of the package.

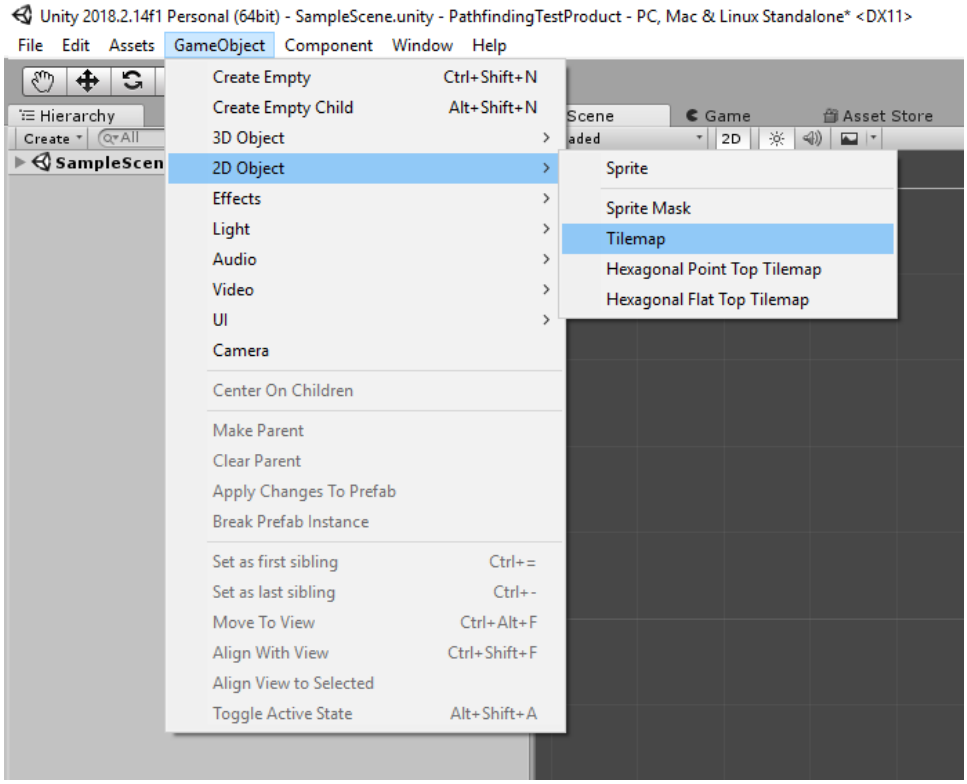
This documentation assumes the reader has some knowledge about the functionality of Unity, particularly *Tilemaps* and *Grids*. It is recommended the reader familiarizes themselves with the current Unity documentation on this functionality before using *Simple Tile Pathfinding*.

Getting Started (Creating a Grid)

Open up Unity.

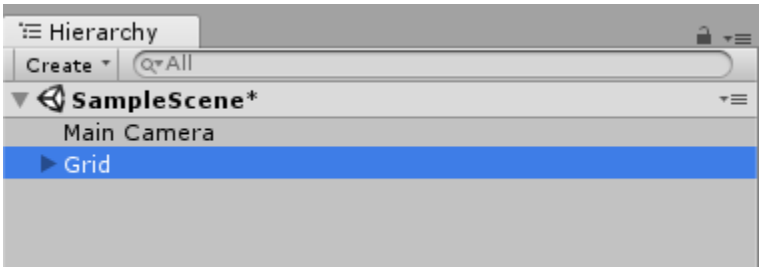
Create a new project in 2D mode (although this isn't a requirement).

Go to the menu and select *GameObject* -> *2D Object* -> Tilemap.



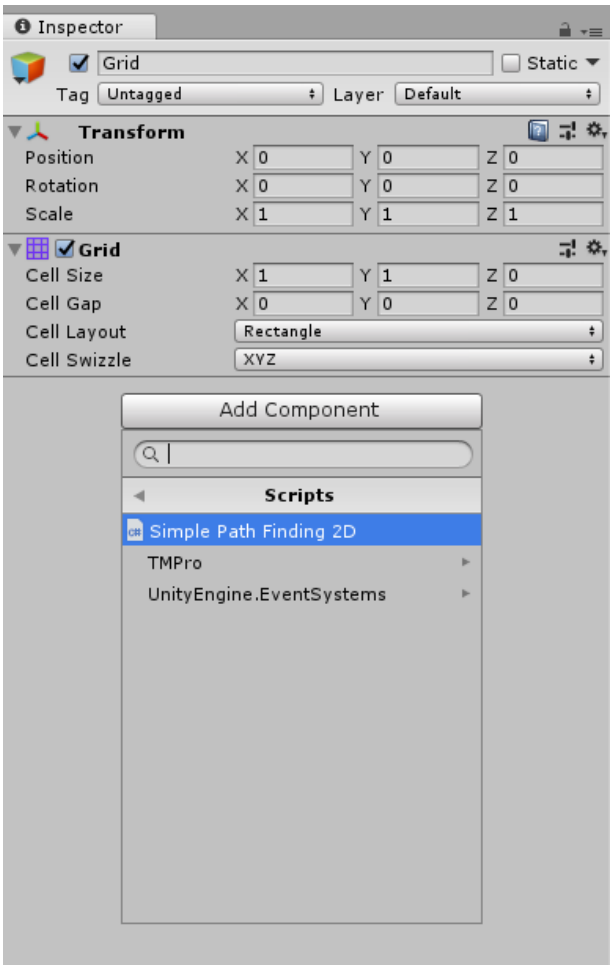
A *GameObject* named Grid should now exist in your scene.

Click on the Grid *GameObject* in the hierarchy view.

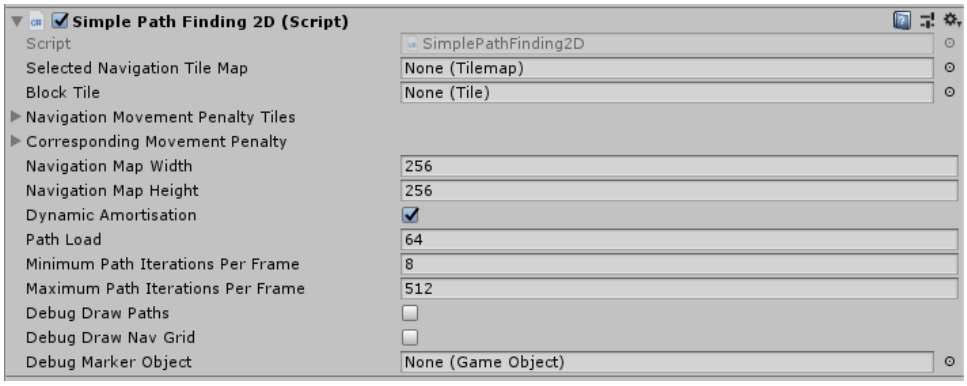


Go to the Inspector and select *Add Component*.

Navigate to *Scripts* and then select the script titled Simple Path Finding 2D.



This script/component is the core of *Simple Tile Pathfinding*. All *Grids* that you intend to perform pathfinding on must have a *SimplePathFinding2D* component attached.



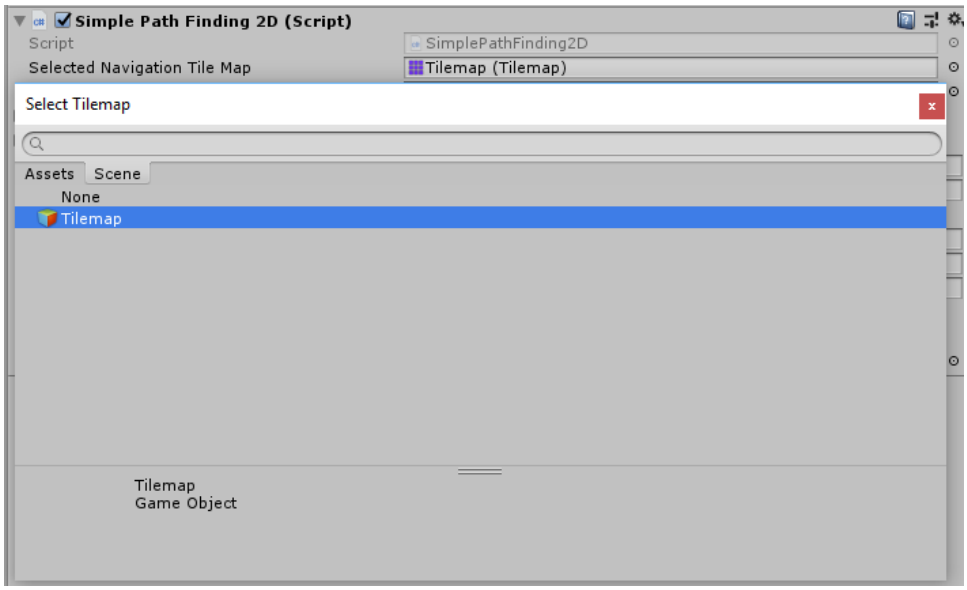
There are a variety of different parameters that can be configured for the *SimplePathFinding2D* component. However, only two are required before we can begin pathfinding:

Selected Navigation Tile Map – This is the *Tilemap* that the *SimplePathFinding2D* component will use as a "mask". I.e. it will use tile cells from this *Tilemap* to create a navigation grid. This will be referred to as the *Navigation Tilemap*.

Block Tile – This is the *Tile* from the Selected Navigation Tile Map that will be used to represent a blocked navigation node in the navigation grid. A blocked node is a node that cannot be traversed. A blocked node could represent something like a wall or obstacle in the game world.

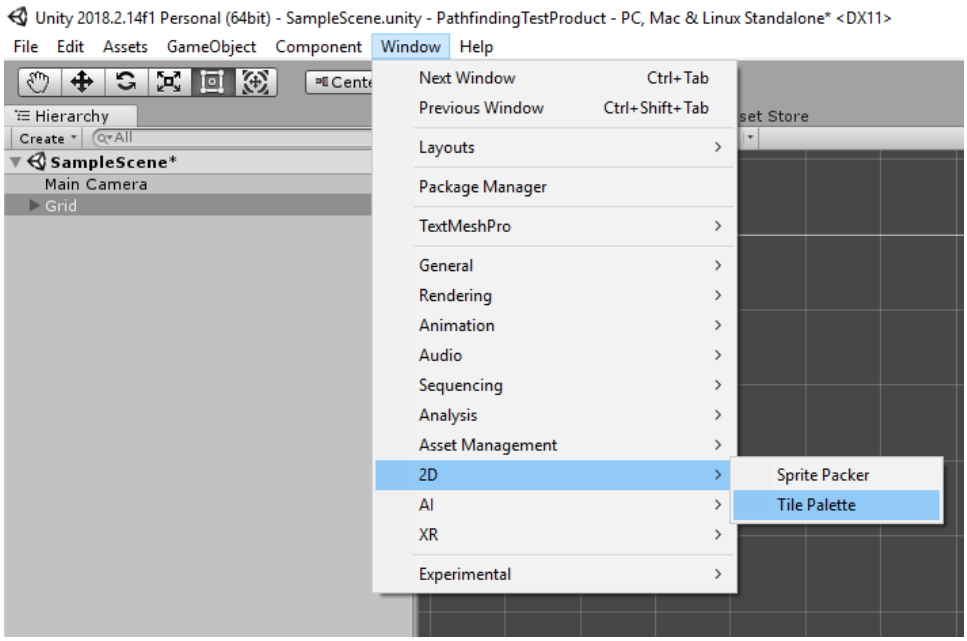
We can ignore the rest of the parameters on this component for now. Just remember that we need to specify the *Navigation Tilemap* AND the *Block Tile* before we can begin any pathfinding.

Select the Selected Navigation Tile Map and select a *Tilemap* of your choosing. (By default, *Grids* are created with a child *Tilemap* named Tilemap)



Now we need to create some tiles to use with our *Navigation Tilemap*.

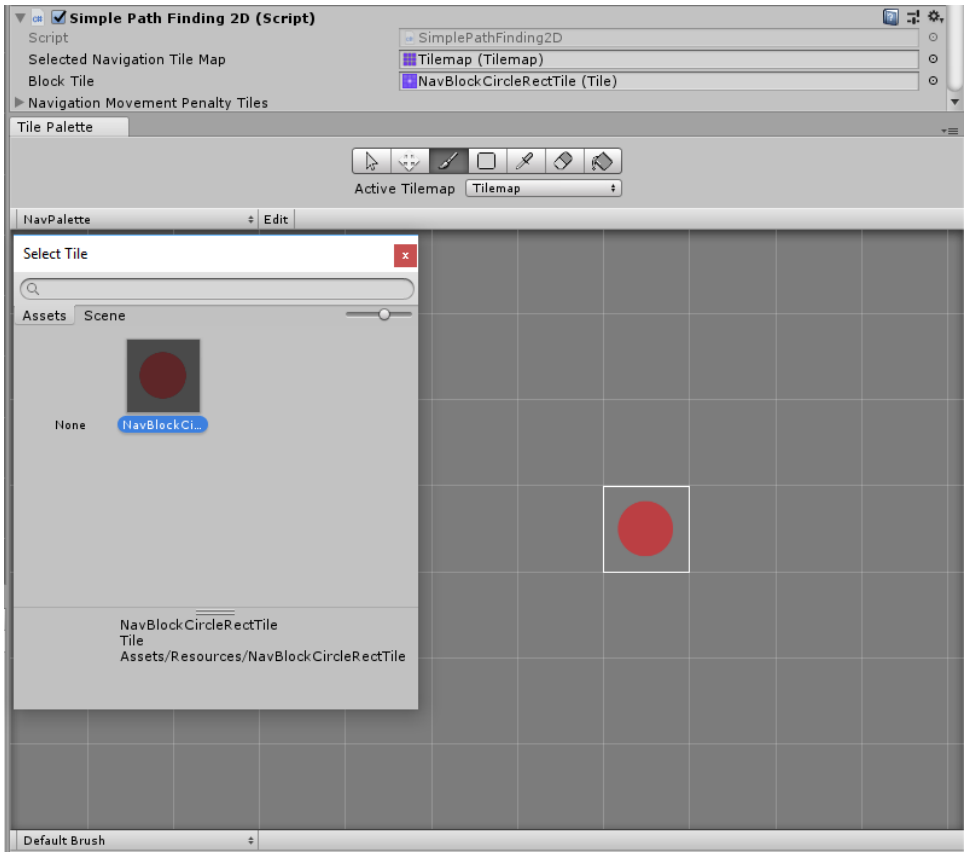
Navigate to *Window -> 2D -> Tile Palette*



In the *Tile Palette* window select *Create New Palette*.

You'll be prompted to create and select a folder which will store your tile assets for this palette.

When you've done this, drag a *Sprite* into your *Tile* palette. This *Sprite* will be used to represent the *Block Tile*.



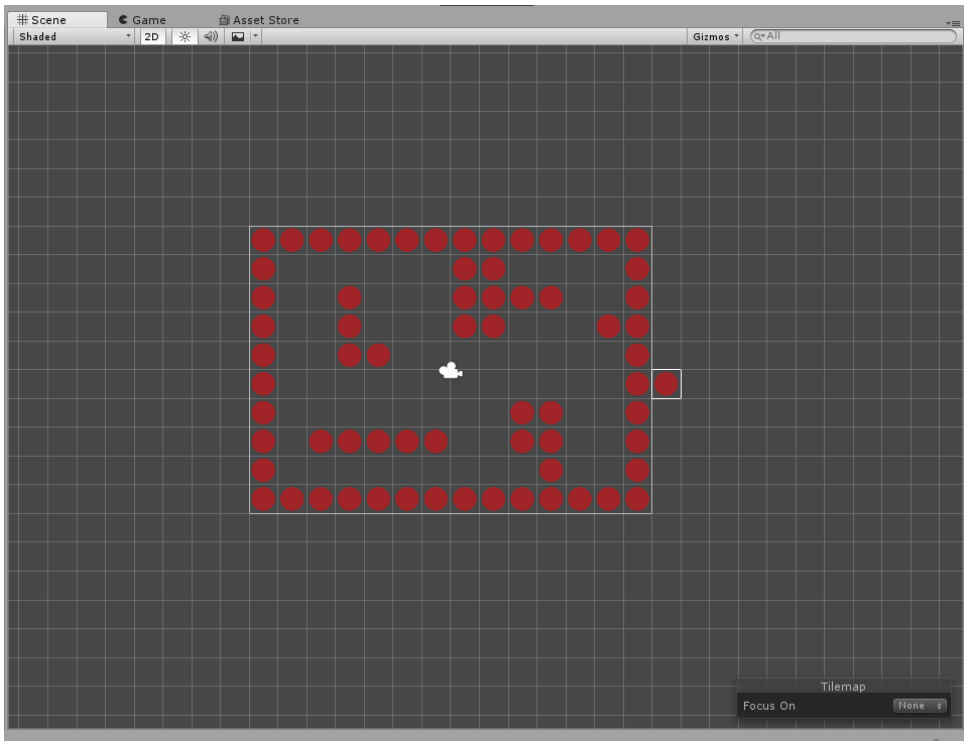
Go back to the *Grid GameObject* and the *Block Tile* parameter of the *SimplePathFinding2D* component. Select this new *Tile* to be the *Block Tile*.

This *Grid* and its *Tilemap* are now configured for pathfinding!

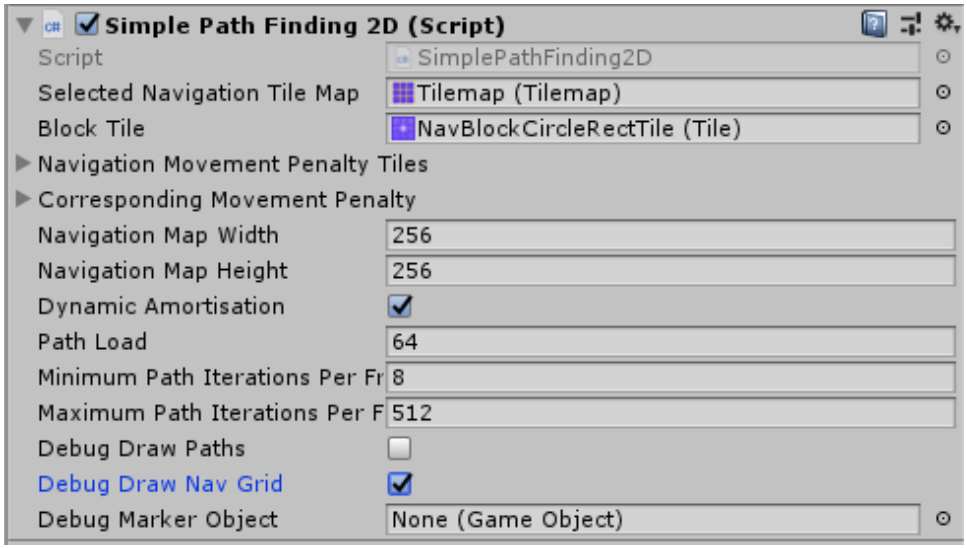
However, there is no tiles on our *Grid*. Our *Tilemap* is empty!

Select the *Tile* in the *Tile Palette* and make sure the *Active Tilemap* is set to the *Navigation Tilemap*.

Start drawing your tiles in the *Scene* view. Make sure you draw a nice maze that will create interesting paths!



If you were to run this *Scene* in Game mode you would notice a blank screen. This is because the *Navigation Tilemap* is hidden from view when the game is running. You can disable this by checking the Debug Draw Nav Grid check box in the *SimplePathFinding2D* component.



This option will display the *Navigation Tilemap*, primarily for debugging purposes.

The *Navigation Tilemap* is used to inform the creation of the navigation grid. It is recommended to put decorative tiles (tiles you want the user to see) on another *Tilemap* that sits on a layer behind or in front of the *Navigation Tilemap*.

Anyway, let's create a simple path.

For now, we'll create a simple component that creates a path between the centre of the grid and to a clicked position on the grid.

Click on the *Grid GameObject* and add a new component. Create a new script. Name it something like MousePath. Open this script file, you should have something like the picture below.

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  public class MousePath : MonoBehaviour {
6
7      // Use this for initialization
8      void Start () {
9
10     }
11
12     // Update is called once per frame
13     void Update () {
14
15     }
16 }
17
```

There is only one object in *Simple Tile Pathfinding* that we really care about. This is the *Path* object. *Path* objects are used to generate paths from A to B on a *Grid*. Create a *Path* object like below.

```
5  public class MousePath : MonoBehaviour {
6
7      private SimplePF2D.Path path;
8
9      // Use this for initialization
10     void Start () {
11
12         SimplePathFinding2D pf = GameObject.Find("Grid").GetComponent<SimplePathFinding2D>();
13         path = new SimplePF2D.Path(pf);
14     }
15
```

Here we have declared a *Path* called path using the namespace *SimplePF2D*. All *Simple Tile Pathfinding* objects exist under this namespace.

We cannot create a *Path* without pointing to a *SimplePathFinding2D* component. Otherwise the *Path* doesn't know what *Grid* to use when navigating.

We find our *Grid GameObject* and we retrieve it's *SimplePathFinding2D* component. We then create a new Path with this component.

Now that this *Path* has been initialized we can use it to create paths across our *Grid*. Look at the example below.

```
// Update is called once per frame
void Update () {

    // When we click the left mouse button
    if (Input.GetMouseButtonDown(0))
    {
        // Get the world position of the user's click
        Vector3 mouseworldpos = Camera.main.ScreenToWorldPoint(Input.mousePosition);
        mouseworldpos.z = 0.0f;

        // Create a path from the 0,0,0 position to the mouseworldpos.
        path.CreatePath(new Vector3(0.0f, 0.0f, 0.0f), mouseworldpos);
    }
}
```

Here we check to see when the left mouse button is pressed within the `Update()` function of this component. We retrieve the world position of this click.

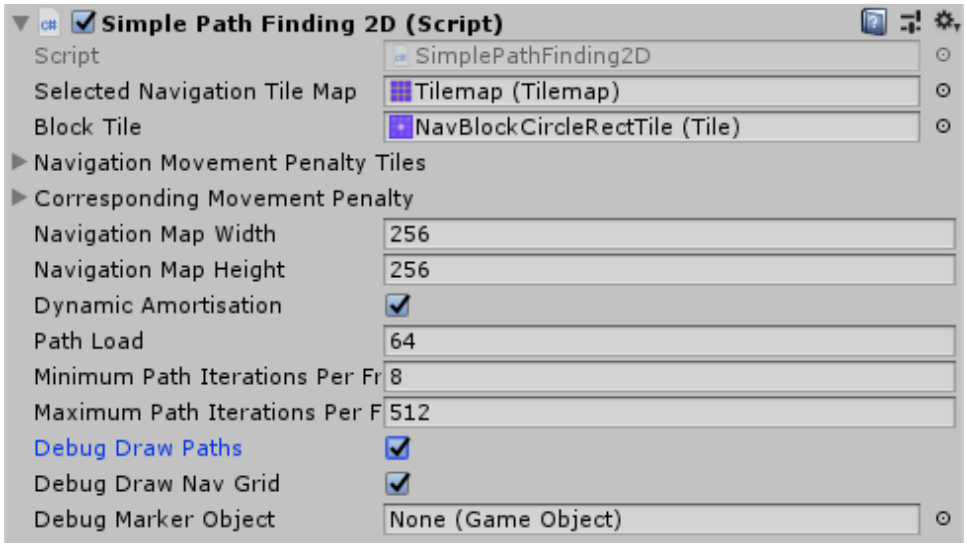
We then use the path object to create a path using the `CreatePath` function.

This function takes a start position and end position and creates a path between them. In this case, we are creating a path between the 0,0,0 position and our mouse click position.

(*Simple Tile Pathfinding* can be used in 3D which is why all world coordinates have 3 dimensions. As long as you have a *Grid* and *Tilemap*, *Simple Tile Pathfinding* can be used)

Go back to the Scene view and run the game. You'll notice that no path is drawn when you click on the *Grid*. This is because we aren't drawing our path.

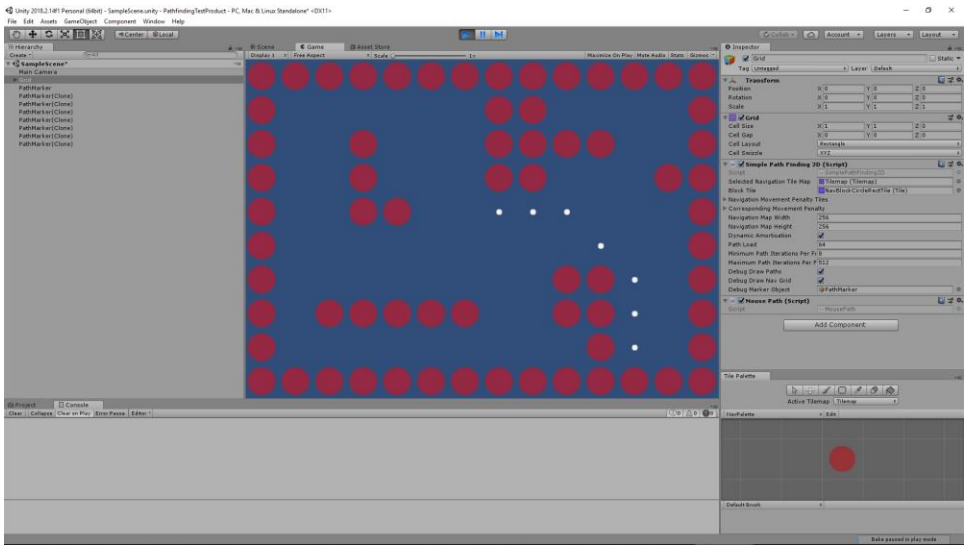
A simple way to draw a path is to check the Debug Draw Paths checkbox on the *SimplePathFinding2D* component.



You will also need to select a Debug Marker Object. This will be a *GameObject* that is instantiated at every point on the path in order to display it. This object can be as simple as something like a *Sprite*.

(It is not recommended to leave the debug options on as they can be a drain on performance).

After selecting a Debug Marker Object and checking Debug Draw Paths run the game and begin clicking.



Wherever you click you will see a generated path. If you don't see a path you may need to unblock the 0,0,0 position, or change the start position of the path.

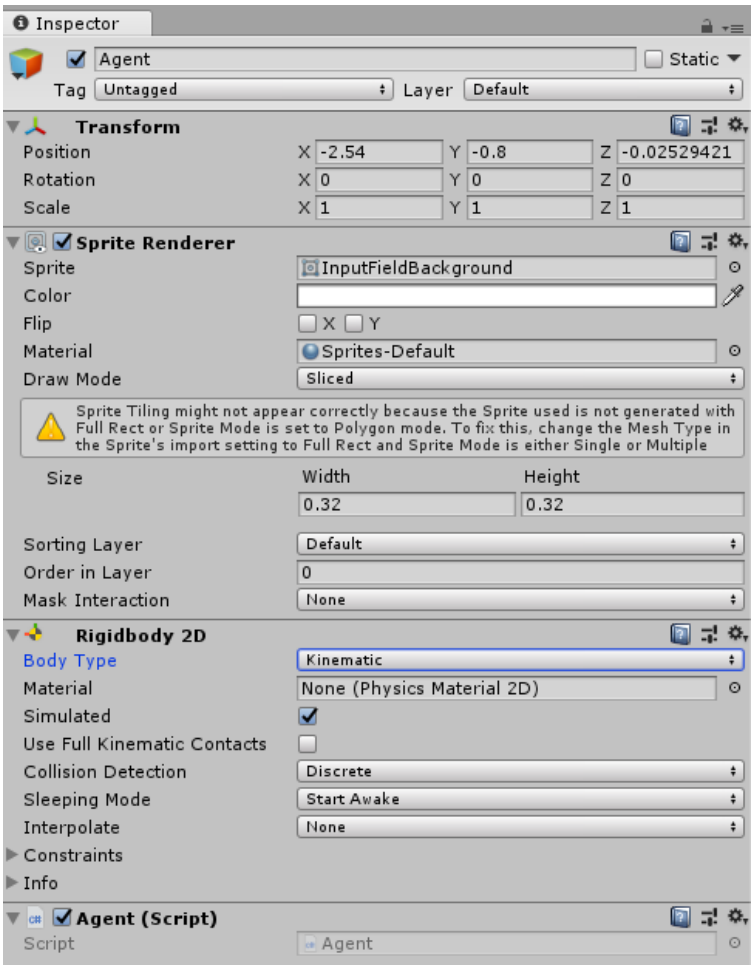
And that's it! This will get you started with the basic and rudimentary, but ultimately important, aspects of *Simple Tile Pathfinding*. Refer to this tutorial when creating any *Grid* you wish to perform pathfinding on.

Using Paths

In any pathfinding system we want agents following and potentially creating a variety of paths. *Simple Tile Pathfinding* doesn't necessarily provide a comprehensive way to control agents via state machines or behavior trees etc etc. However, its *Path* objects are designed to be adaptable, extensible and easily useable with any of these systems. The following tutorial will give an example on how to do this.

Assuming we have completed the previous tutorial, we currently have a Scene with a maze like *Grid* and a *Navigation Tilemap*, along with a MousePath component creating path's when we click the mouse. You should begin by first removing this MousePath component.

We should create a new *GameObject* called *Agent* that has a *Sprite* component and a *Rigidbody2D* component (Set its *Body Type* to *Kinematic*). Give this *Agent* a nice distinguishable *Sprite* image.



Create a new Script on this *Agent* called Agent. Open this Script in an IDE of your choice.

We want this *Agent* to create paths on the fly and then follow these paths to their destination.

We'll start by initialising the variables of this component in its `Start()` routine.

```
5   public class Agent : MonoBehaviour {
6
7       private SimplePF2D.Path path;
8       private Rigidbody2D rb;
9       private Vector3 nextPoint;
10      private float speed = 5.0f;
11      private bool isStationary = true;
12
13      // Use this for initialization
14      void Start () {
15
16          // Create our new paths using the SimplePathFinding2D object attached to a Grid GameObject.
17          path = new SimplePF2D.Path(GameObject.Find("Grid").GetComponent<SimplePathFinding2D>());
18          rb = GetComponent<Rigidbody2D>();
19          nextPoint = Vector3.zero;
20      }
```

Here we can see we have declared a variety of private members within this component. We have a reference to a *Path* that it will follow, a reference to the *Rigidbody2D* component on this *Agent's*

GameObject and a vector called `nextPoint`. The variable `nextPoint` will hold the value of the next point in the path that we generate.

```
// Update is called once per frame
void Update () {

    if (Input.GetMouseButtonDown(0))
    {
        Vector3 mouseworldpos = Camera.main.ScreenToWorldPoint(Input.mousePosition);
        mouseworldpos.z = 0.0f;
        path.CreatePath(transform.position, mouseworldpos);
    }
```

Again, as in the previous tutorial, within the `Update()` function we create a path when the user clicks. Except this time the path begins at the *Agent's* current position.

```
// Update is called once per frame
void Update () {

    if (Input.GetMouseButtonDown(0))
    {
        Vector3 mouseworldpos = Camera.main.ScreenToWorldPoint(Input.mousePosition);
        mouseworldpos.z = 0.0f;
        path.CreatePath(transform.position, mouseworldpos);
    }

    if (path.IsGenerated())
    {
```

The next step is important. The line `path.IsGenerated()` checks to see if the path has been successfully created. The creation of *Paths* in *Simple Tile Pathfinding* is spread out over a number of frames. It is amortized. Therefore we have to check if a path has finished processing before we can use it.

```
27      if (path.IsGenerated())
28      {
29          if (isStationary)
30          {
31              // Get the next point in the path as a world position. Returns true when this is successful.
32              if (path.GetNextPoint(ref nextPoint))
33              {
34                  rb.velocity = nextPoint - transform.position;
35                  rb.velocity = rb.velocity.normalized;
36                  rb.velocity *= speed;
37                  isStationary = false;
38              }
39              else // When GetNextPoint fails it means we have reached the end of the path.
40              {
41                  rb.velocity = Vector3.zero;
42                  isStationary = true;
43              }
44          }
45          else
46          {
47              Vector3 delta = nextPoint - transform.position;
48              if (delta.magnitude <= 0.2f)
49              {
50                  rb.velocity = Vector3.zero;
51                  isStationary = true;
52              }
53          }
54      }
55      else
56      {
57          rb.velocity = Vector3.zero;
58          isStationary = true;
59      }
```

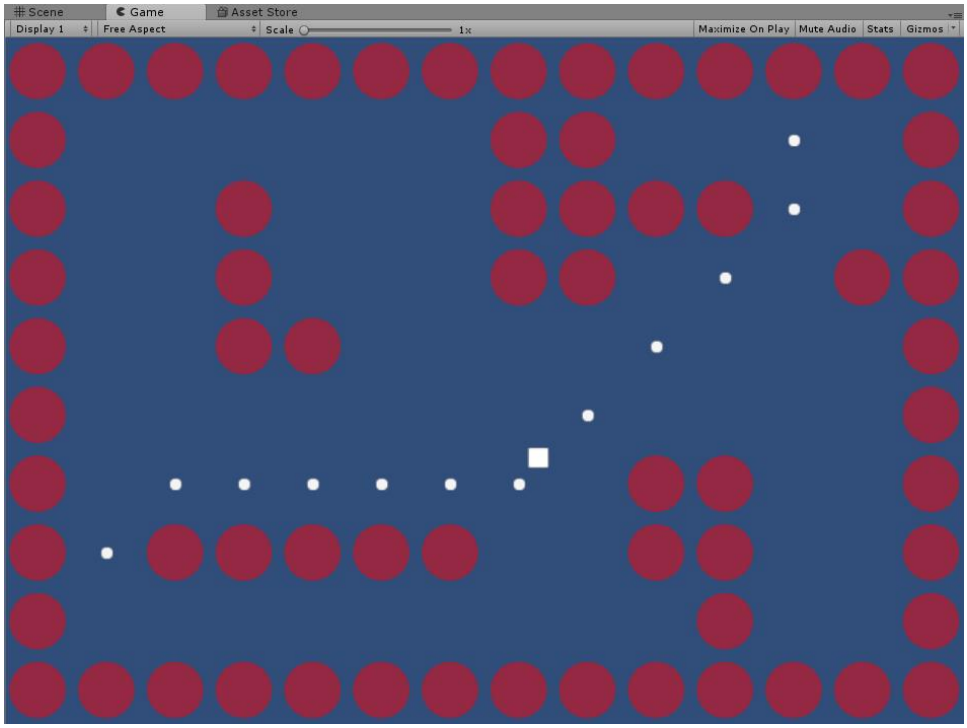
The member variable `isStationary` tracks the state of our *Agent*. We do a check to see if the *Agent* is in this state. If the *Agent* is, then we retrieve the next point in our path.

```
// Get the next point in the path as a world position. Returns true when this is successful.
if (path.GetNextPoint(ref nextPoint))
{
    rb.velocity = nextPoint - transform.position;
    rb.velocity = rb.velocity.normalized;
    rb.velocity *= speed;
    isStationary = false;
}
else // When GetNextPoint fails it means we have reached the end of the path.
{
    rb.velocity = Vector3.zero;
    isStationary = true;
}
```

All *Path* objects have an internal pointer that keeps track of the current point that the *Path* object is on. When a *path* is created this pointer starts at the beginning of the path. With each call of `GetNextPoint()` this pointer is incremented and the next point of the path is returned by reference. When `GetNextPoint()` returns false, this means the internal pointer has reached the end of the path.

The rest of the above code involves moving the *Agent* to the `nextPoint`. Stopping within a certain radius, changing state and then retrieving the next point in the path until all those points in the path are exhausted.

If we run Unity in Game Mode we can see that we can direct our *Agent* around the *Grid*.



Although relatively rudimentary, this the basics of using *Paths* in *Simple Tile Pathfinding*.

Movement Penalty Tiles

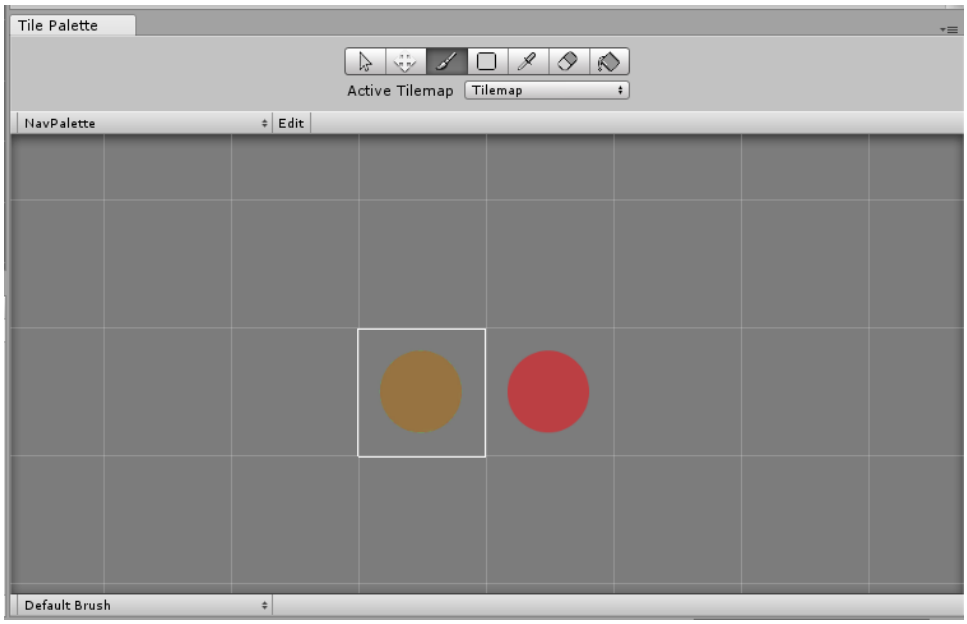
Sometimes we want to encode additional information into our *Navigation Tilemap*.

We may want to indicate to the pathfinding system that some tiles/navigation cells are slower than others.

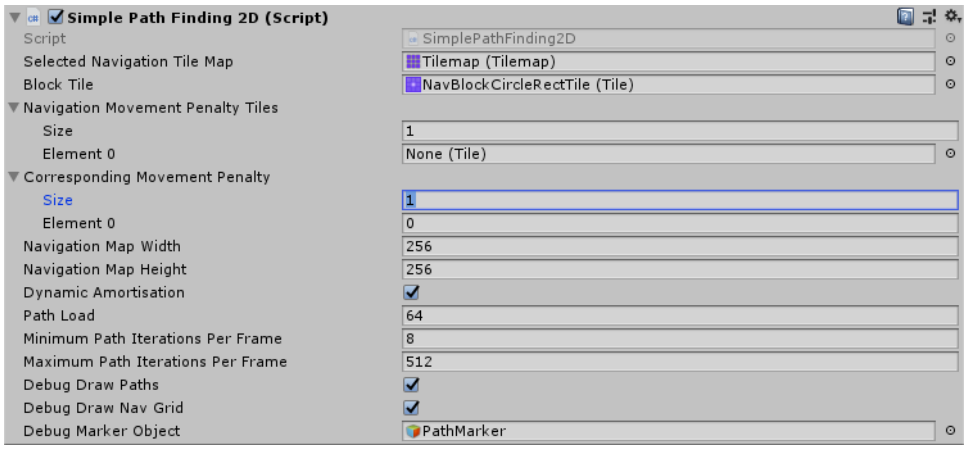
From a gameplay perspective, perhaps some tiles are coated in honey while other tiles are not. These honey tiles should therefore have intrinsic movement penalties that the pathfinding system should account for. They are still traversable tiles, but they are a slower option.

We can create additional tiles in our *Navigation Tilemap* and specify their movement penalty to the *SimplePathFinding2D* component.

Create a new tile to be used by the *Navigation Tilemap*. Make sure it has a distinguishable *Sprite* from the *Block Tile* used by the same *Navigation Tilemap*.

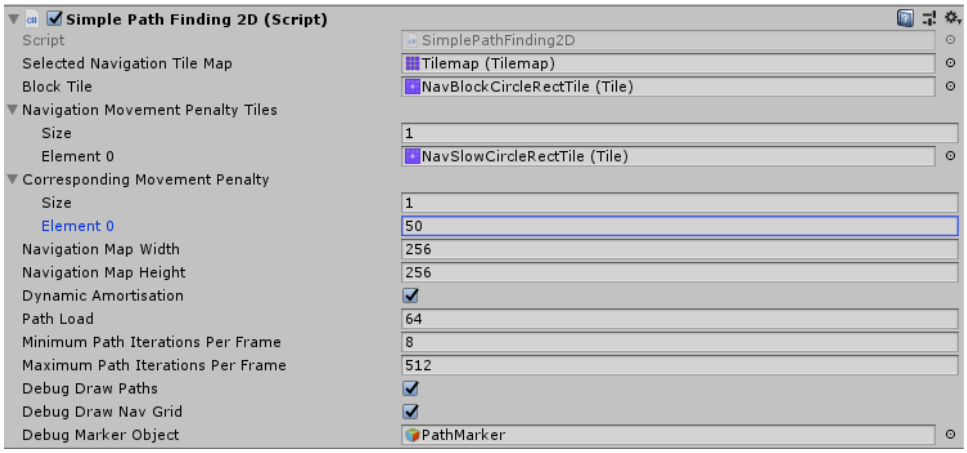


Now that we have a new tile, we can go into the *SimplePathFinding2D* component on the *Grid*. Select the Navigation Movement Penalty Tiles parameter and set its size to 1. Then do the same for the Corresponding Movement Penalty.

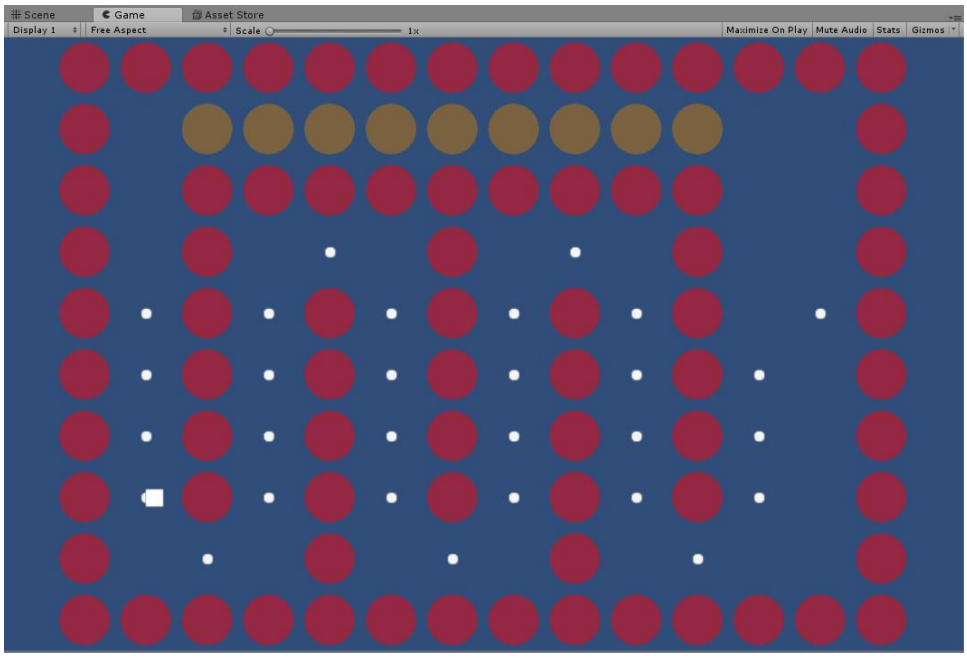


The Navigation Movement Penalty Tiles parameter is an optional list of tiles that specify the movement penalty wherever that tile appears in the *Navigation Tilemap*. The value of the movement penalty is specified in the Corresponding Movement Penalty list.

Add the newly created tile to the list and set its corresponding movement penalty to 50.

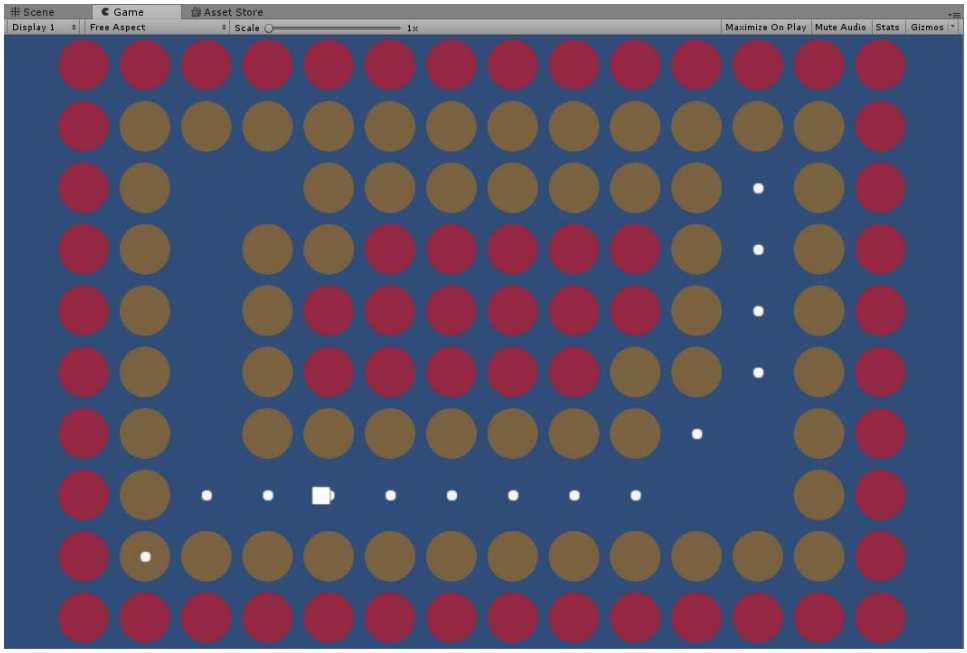


Now we can start adding these new tiles to the game world. Create a maze where our *Agent* in the previous tutorial has a choice between a short path or a long path, but the short path is coated in the "honey" tiles.



As you can see above, the *Agent* decides to take the longer path due to the huge penalty cost of the shorter path thanks to the introduction of our new tile.

Using movement penalties like this can be useful if you don't want your Agents hugging walls (as that is often the shortest path). As seen below, you can place these "honey" tiles at wall edges and adjust their penalty accordingly.



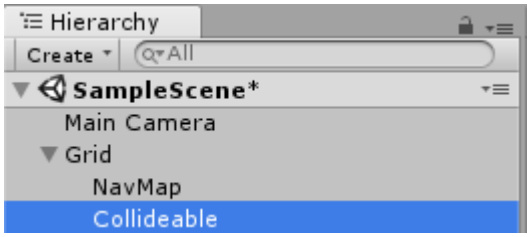
Collideable Tilemaps

Drawing the *Navigation Tilemap* by hand can be tedious, boring and error prone. In *Simple Tile Pathfinding* the underlying *Navigation Grid* can be changed during runtime or during initialisation dynamically so that we don't have to do this.

Let's say we have a separate *Tilemap* that represents areas that we do not want to be traversable. For example, a *Tilemap* that could represent walls in your Game world.

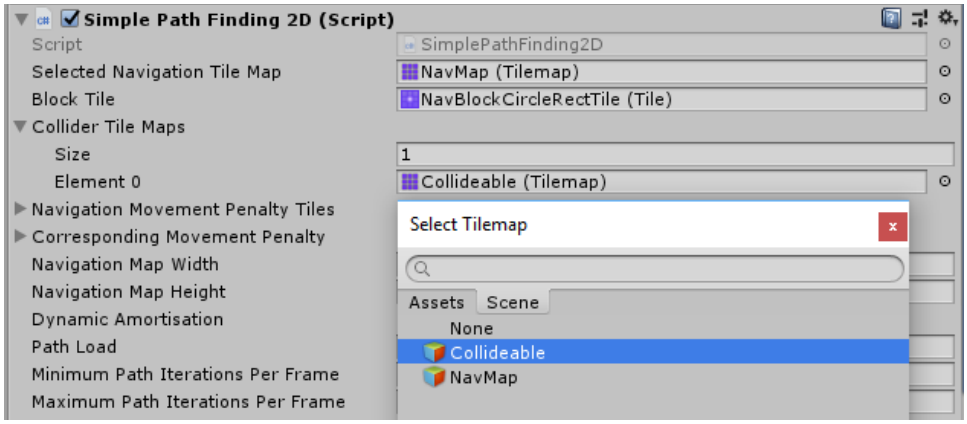
We can use this *Tilemap* to define the navigation space of our *Grid*.

Create a *Grid* with two *Tilemaps*. One *Tilemap* will be our *Navigation Tilemap*. We'll call this NavMap. The other *Tilemap* we will call Collideable.



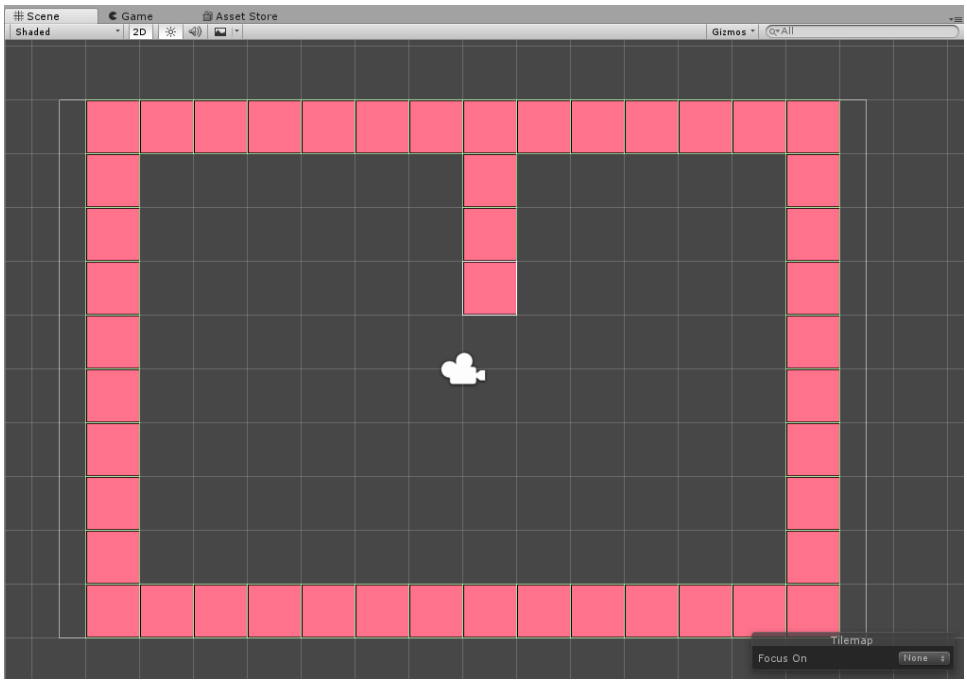
Add a *SimplePathFinding2D* component to your *Grid GameObject* and set its *Navigation Tilemap* to NavMap. (Create a *Block Tile* and set this up accordingly)

In the inspector for the *SimplePathFinding2D* component on the *Grid GameObject*, add the Collideable *Tilemap* to the Collider Tile Maps list. Shown below.

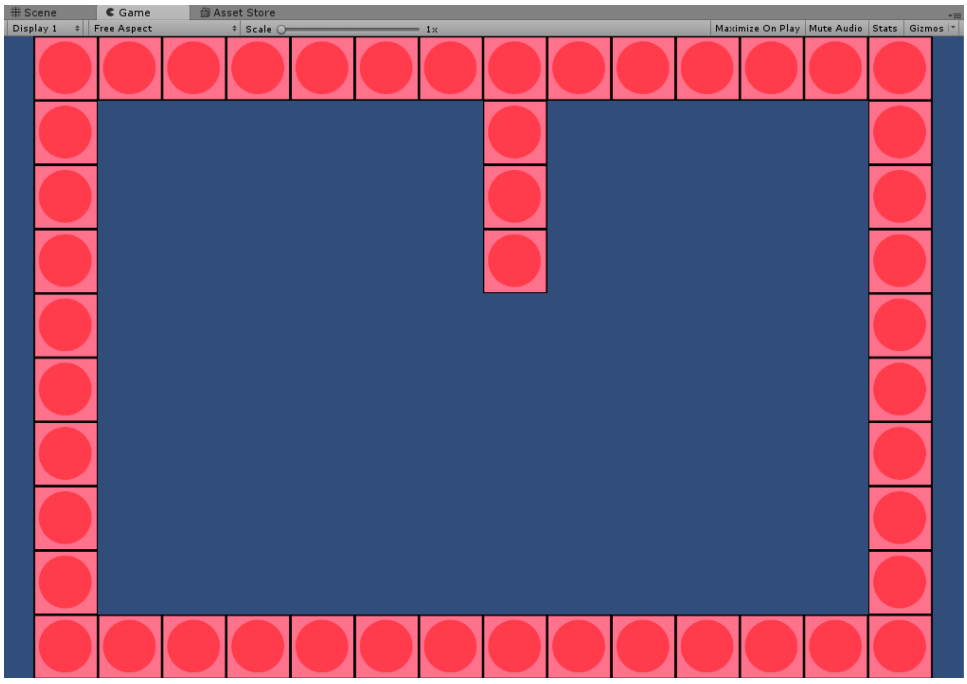


Now, whenever you draw tiles on the Collideable *Tilemap*, this will block tiles on the *Navigation Tilemap*.

Try drawing some tiles on the Collideable *Tilemap* .



The above image displays a series of pink tiles that represent walls, that have been drawn on the Collideable *Tilemap*. If we select Debug Draw Nav Grid on the *SimplePathFinding2D* component of the *Grid GameObject* and run Unity in Game mode we can see...



That these tiles are no longer traversable. (You may need to set the sort order of the NavMap *Tilemap* to be above that of the Collideable *Tilemap* in order to see the *Navigation Tilemap*).

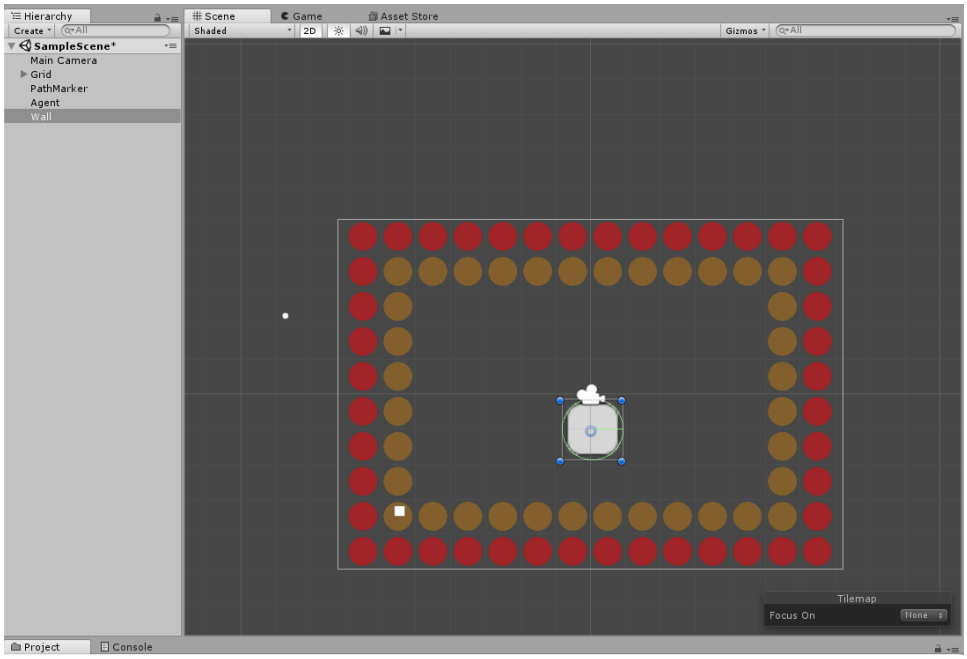
We can set any number of *Collideable Tilemaps* in the *SimplePathFinding2D* component. You may have various layers of walls/obstacles/buildings that you do not want to be traversable.

Dynamic Navigation Grid Generation

We may have an object in our Game world that wants to change our navigation grid at run time. We can do this programmatically.

Create a *Grid* with a *Tilemap* and a *SimplePathfinding2D* component.

Create a rectangular *Sprite GameObject* and call it Wall. Give this object a *CircleCollider2D* component. Make it a substantial size so that it at least covers more than one *Grid* cell.



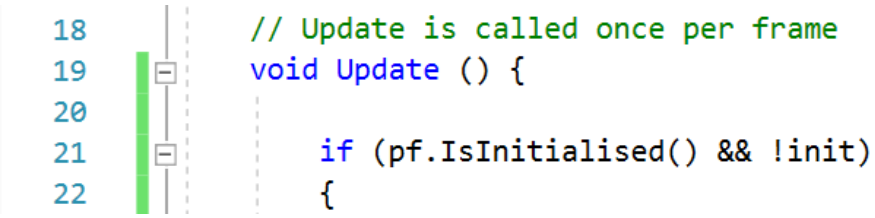
In the above image we can see the created *Sprite* called Wall. It sits in the middle of the *Grid*. We could quite easily draw on the *Navigation Tilemap* by hand and block the tiles underneath the wall object manually. However, what if we decided to move the wall object to a different position? We would have to draw on the *Navigation Tilemap* once again.

Create a new Script on the *Wall* object called Wall. We are going to use the *Bounds* of the collider component to prevent this object from being traversable.

```
5   public class Wall : MonoBehaviour {
6
7       private SimplePathFinding2D pf;
8       private Collider2D c;
9       private bool init = false;
10
11      // Use this for initialization
12      void Start () {
13
14          pf = GameObject.Find("Grid").GetComponent<SimplePathFinding2D>();
15          c = GetComponent<CircleCollider2D>();
16      }
```

Here we do some simple initialisation. We retrieve the *Grid's SimplePathFinding2D* component and then the collider component of this object.

Something to note is that we have to be careful accessing members of the *SimplePathFinding2D* component in the `Start()` function of another component. This is because the *SimplePathFinding2D* component might not initialise until AFTER this particular `Start()` function is called. Therefore, we need to check if the pf object in the above example is initialised.



```
18      // Update is called once per frame
19      void Update () {
20
21          if (pf.IsInitialised() && !init)
22          {
```

We do this in the `Update()` function. We use a boolean flag called `init` to ensure this only happens once.

```
18      // Update is called once per frame
19      void Update () {
20
21          if (pf.IsInitialised() && !init)
22          {
23              Vector3Int min = Vector3Int.zero;
24              Vector3Int max = Vector3Int.zero;
25              bool successFlag = pf.GetNavNodesInBounds(c.bounds, ref min, ref max);
```

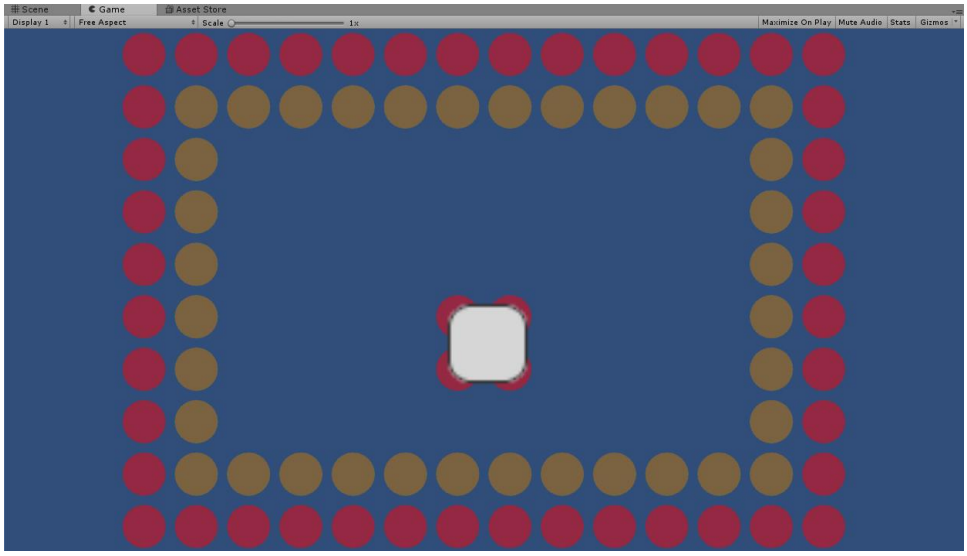
Now we call a member function on the `pf` object called `GetNavNodesInBounds`. This function takes the bounds of the collider component and returns by reference the min and max positions of the navigation nodes that are contained within it.

We can cycle through these min and max coordinates and manipulate the navigation nodes that sit within this area.

```
if (pf.IsInitialised() && !init)
{
    Vector3Int min = Vector3Int.zero;
    Vector3Int max = Vector3Int.zero;
    bool successFlag = pf.GetNavNodesInBounds(c.bounds, ref min, ref max);

    if (successFlag)
    {
        for (int x = min.x; x < max.x; x++)
        {
            for (int y = min.y; y < max.y; y++)
            {
                SimplePF2D.NavNode node = pf.GetNode(x, y);
                node.SetBlocked(true);
            }
        }
    }
    init = true;
}
```

In the above code snippet we iterate through each *NavNode* in this bounding area. We retrieve a node at a specified position using the `GetNode` function. We then specify that this node should be blocked I.e. not traversable. Try running Unity in Game mode now.



We can see that the nodes underneath our *Wall* object have now been blocked and are not traversable. (Make sure you have Debug Draw Nav Grid checked on the correct *SimplePathFinding2D* component so that you can see the nodes).

You can do this for any form of collider that has a bounding box.

Multiple Grids

Sometimes, with maps that have very large, empty spaces it may be useful, for the purposes of performance, to have a *Grid* with a very large cell size. This allows quick path generation over a large area.

When the map becomes very detailed (or perhaps the pathfinding *Agent* enters the screenspace) we can defer to a *Grid* with a much smaller cell size. This requires the use of multiple *Grids*.

Simple Tile Pathfinding supports this. With each *Grid* object we can have a new *SimplePathFinding2D* component. We can create unique paths for each one of these *SimplePathFinding2D* components.

```
SimplePF2D.Path largePath;
SimplePF2D.Path smallPath;
// Create two different path objects that navigate on two seperate grids.
largePath = new SimplePF2D.Path(GameObject.Find("LargeGrid").GetComponent<SimplePathFinding2D>());
smallPath = new SimplePF2D.Path(GameObject.Find("SmallGrid").GetComponent<SimplePathFinding2D>());
```

In the above code snippet we isolate two separate *Grid* objects and their respective *SimplePathFinding2D* components. We then create a small and large path using these components.

We can create any number of *Grids* that can be navigated around with separate paths. The only limitation is memory and/or CPU usage and your ability to keep track of multiple grids at the same time.

Performance

CPU Cycles

Although the aim of *Simple Tile Pathfinding* is ease of use, significant effort has been placed on generating paths quickly and efficiently.

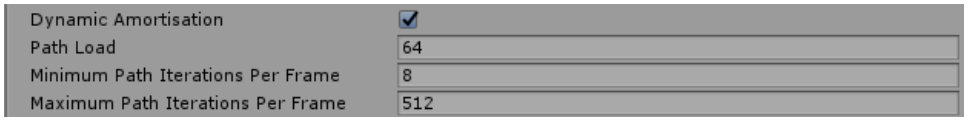
Pathfinding can be particularly performance heavy depending on the number of paths being found and the size of the pathfinding search area. The first thing to note is that *Simple Tile Pathfinding* is not multithreaded, however this doesn't mean it is entirely useless. It provides a number of options to mitigate performance issues.

Path generation in *Simple Tile Pathfinding* is amortized. They are generated over a number of frames until completed. This needs to be taken into account when creating and using paths. A path will not be ready on the same frame that it is created (unless the path is very small!).

Multiple paths are created concurrently, with the first created path being prioritized.

If multiple paths are being created at the same time then we may want to regulate how many paths are processed per frame and how many iterations each process performs.

If we view the *SimplePathFinding2D* component we can see a number of options.



Path Load is the number of paths that are processed concurrently per frame. Reducing this number will increase performance. However, it means that path generation is spread out over more frames. Therefore, it will take more real time to generate all the paths.

Path iterations is the amount of iterations the A* algorithm will perform for each path per frame. If we have n paths and x iterations we will perform n multiplied by x path iterations per frame. Reducing this value will improve performance, but it means that it will take more real time to generate the paths.

The Dynamic Amortisation option means that *Simple Tile Pathfinding* will linearly scale between the minimum and maximum iterations per frame based on the number of paths being processed.

If the number of paths being processed is equal to the Path Load value then *Simple Tile Pathfinding* will use the Minimum Path Iterations Per Frame value. If the number of paths being processed is equal to 1, then the Maximum Path Iterations Per Frame value will be used instead.

Turning this option off means that *Simple Tile Pathfinding* does not scale the iterations per frame and defaults to the maximum path iterations per frame value.

These settings apply per *Grid*. They are not global settings.

We can programmatically override the number of iterations a path performs. When creating a *Path* we can set an optional parameter.

```
path.CreatePath(transform.position, mouseworldpos, true, true, 8);
```

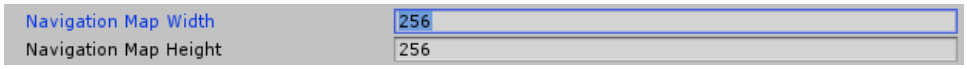
The last parameter here determines that the path should use 8 iterations per frame when processing. If we set it to zero, or omit it completely, then it will default to the iterations determined by the *SimplePathFinding2D* component.

If the search area of your *Grid* is very large this may also decrease performance. If this *Grid* is mostly made of empty space, consider creating a new *Grid* with a larger tile size and lower granularity. We can then defer to a smaller *Grid* to perform more accurate pathfinding over a smaller distance when the need arises.

Memory Consumption

We can change the amount of memory that *Simple Tile Pathfinding* uses. Although long paths will inevitably consume more memory, we have a small set of options as to how much memory we allow to be utilised.

All navigation grids have a centre at the 0,0,0 coordinate. We can shrink the width and height of this grid using the options found on the *SimplePathFinding2D* component attached to a *Grid GameObject*.



This reduces the amount of navigation nodes created and thus the amount of memory used, but it does decrease the size of the available navigation grid.

There are also a number of hard coded options we can change to reduce our memory footprint.

Within the *AStarSearch* class we have a static list called `freeList`. This will grow over time as more paths are created. This `freeList` is capped at a max size set by a parameter called `freeListMaxSize`. Increasing this value means that we trade more memory for an increase in performance. Decreasing this value does the opposite.

The *AStarSearch* class also has a linked list called `openBins`. This is a list of arrays that is used to sort navigation nodes by their F values (used by A*). The number of these `openBins` and the number of elements they contain is determined by two values. These values are `maxopenbinF` and `maxbins`. Increasing both these values increases memory usage and also increases performance. Decreasing these values does the opposite, decreasing memory usage and also decreasing performance.

NavNode

Description:

Stores data about individual cell/nodes in a navigation grid. Loosely associated with *Tiles*, this object sits "underneath" them where one *NavNode* corresponds to one *Tile*. Has it's own coordinates that index directly into the navigation grid array of a *SimplePathFinding2D* component. Stores information about whether the tile it corresponds to is traversable and/or has a movement penalty.

Properties:

pf	The SimplePathFinding2D component this NavNode is associated with
isBlockedFlag	A flag that indicates whether this node is traversable or not
index	The position in the Navigation Grid that this node exists at
movementpenalty	A movement penalty associated with the node. Useful if you want a node to be traversable but not desirable when pathfinding

Methods:

NavNode(SimplePathFinding2D newpf)	Constructor. Requires a SimplePathFinding2D component
void SetIndex(Vector3Int newIndex)	Sets the position in the Navigation Grid that this node exists at
Vector3Int GetIndex()	Returns the Navigation Grid index/position of this node
void SetBlocked(bool newIsBlockedFlag)	Sets the isBlockedFlag which signifies whether the node is traversable or not
bool IsBlocked()	Returns the flag that signifies that this node is traversable
bool IsIgnorable()	Returns the whether the node is blocked or not
void Reset()	Resets the movement penalty to zero
static int DirectionalGCost(DiagonalDirectionEnum dir)	Returns a cost value based on the direction. Diagonals "cost" more than orthogonals
int GetMovementPenalty()	Returns the value of the movement penalty

void SetMovementPenalty(int val, UnityEngine.Tilemaps.TileBase tile = null)	Sets the value of the movement penalty. In debug mode, sets the tile at this node position.

Path

Description:

A class that stores and traverses a path generated by an *AStarSearch* object.

Properties:

pf	The object that contains the navigation grid that this path will use to search.
pathpoints	A list of vectors in tile coordinates that make up the path. The start of the list is the start of the path.
aStarSearch	An object that handles the A* search.
startpos	Holds the value of the start world position of this path
endpos	Holds the value of the end world position of this path
searchWithDiagonals	Indicates that this path should search using diagonal neighbours
isdynamicflag	Indicates whether the path is dynamic or not. When a path point that is returned via a get function is blocked then the path is recreated. Dynamic paths regenerate the path from the point of blockage to the end goal position. Static paths regenerate a path from the initially specified start position to the end position.
internalindex	A index of the pathpoint list that tracks the current path point of this path
init	A boolean indicating whether this path is initialised or not

Methods:

Path(SimplePathFinding2D newpf)	Constructor. Requires a valid SimplePathFinding2D component

void Reset()	Resets the internal index to zero and clears the path point list
Void SetPathPointIndex(int index)	Manually sets the internal index to a given index. Clamps this value in range with the path point list
Void SetPathStart()	Sets the internal path point index to zero
void SetPathEnd()	Sets the internal path point index to the end of the path point list
List<Vector3Int> GetPathPointList()	Returns the path point list
int GetInternalPathPointIndex()	Returns the value of the internal path point index
Vector3Int GetPathPoint()	Returns the tile coordinate of the path point pointed to by the internal index. Returns a zeroed vector if the index is out of range
Vector3Int GetPathPoint(int index)	Returns the tile coordinate of the path point pointed to by the specified index. Returns a zeroed vector if the index is out of range
Vector3 GetPathPointWorld()	Returns the world coordinate of the path point pointed to by the internal index. Returns a zeroed vector if the index is out of range.
Vector3 GetPathPointWorld(int index)	Returns the world coordinate of the path point pointed to by the specified index. Returns a zeroed vector if the index is out of range.
bool IsGenerated()	Returns true if the path point list count is greater than 0 and that the aStarSearch object is not in a failed state
bool GetNextPointIndex(ref Vector3 nextPoint, ref int index)	Gets the next point in the path point list from the passed in index value. Returns both the new index and the path point (as a world coordinate) by reference. If the next point is blocked, a new path will be created, resetting this one. Returns false if it reaches the end of the list. Returns true if there was a next point.
bool GetNextPoint(ref Vector3 nextPoint)	Same as above, but uses the internal index of the path
bool GetPreviousPointIndex(ref Vector3 previousPoint, ref int index)	Same as above but the previous point is retrieved instead. Returns false when we are back to the beginning of the list.

bool GetPreviousPoint(ref Vector3 previousPoint)	Same as above, but uses the internal index of the path
bool CreatePath(Vector3 startPosWorld, Vector3 endPosWorld, bool searchUsesDiagonals = true, bool isdynamic = true, int overrideiterations = 0)	Creates a path from one world coordinate to another. Has optional parameters that inform the path as to whether it should search for diagonal neighbours, is a dynamic path and whether it should override the number of iterations it performs per frame
bool CreatePath(Vector3Int startTile, Vector3Int endTile, bool searchUsesDiagonals = true, bool isdynamic = true, int overrideiterations = 0)	Same as above but uses Tile coordinates instead.

AStarSearch

Description:

A class that deals with calculating a path from one position to another on a *Tilemap*. There can be multiple instances of this object. Each A* object progresses through a specified amount of the path finding algorithm per frame (amortised). This means pathfinding is processed concurrently. The amount of "work"/iterations performed by each of these objects can be specified in the *SimplePathFinding2D* component that it is assigned to. This class must only be created alongside a path and only works if there is a valid *SimplePathFinding2D* component.

Properties:

pf	The object that contains the navigation grid
navNodeWrapperMap	A dictionary that maps nav nodes to a nav wrapper class. There is a one to one mapping.
freeList	A pool of available nav wrappers to be used instead of recreated. Shared by all AStarSearch objects.
freeListMaxSize	The max size of the nav wrapper free list/pool. Increasing this increases memory usage
initFlag	Indicates that this class has been initialised
openBins	The "open list". A list of listed searched nodes arranged in ascending order based on their F value.
currentNavWrapper	A nav wrapper object that points to the current nav wrapper being processed

endWrapper	A wrapper that contains the final node in the path
layout	Caches the layout of the Grid (Hexagonal, Rectangular etc)
amortizeNumber	Keeps track of the number of iterations that this object has run per frame
overridedMaxAmortizevalue	Overrides the max amortize value set by the SimplePathFinding2D object
useDiagonals	A flag that signifies that the node grid should be traversed using diagonal neighbours.
maxopenbinF	The max value of the open list bins. Any value above this gets put into the final bin
maxbins	The number of open list bins. The bigger the number the more memory, but the quicker the traversal
isProcessingFlag	A flag that indicates that this object is still processing
hasFailedFlag	A flag indicating this AStarSearch object has failed to find a path

Methods:

AStarSearch(SimplePathFinding2D newpf)	Constructor. Requires a valid SimplePathFinding2D component
NavWrapper CreateNavWrapper(NavNode node)	Creates a nav wrapper that stores temporary information about a nav node for use in the path finding algorithm. These are grabbed from a pool if they are free to use. If not, a new one is created. Only one nav wrapper can be associated with one node
void AddToOpenList(NavWrapper navWrapper)	Adds a nav wrapper to the open list bins based on its F value.
void AddToCloseList(NavWrapper navWrapper)	The closed list is "virtual". Adding to the "closed list" just removes, a nav wrapper from the open list and sets its closed flag to true.
NavWrapper OpenListLookForLowestF()	Searches the open list bins for the nav wrapper with the lowest f value
int CalculateHScore(NavNode currentNode)	Calculates the manhatten distance from the current node to the end node
void CheckAdjacent(NavWrapper currentNavWrapper, DiagonalDirectionEnum dir)	Takes a look at a neighbour node based on a set direction. Checks if this node is open and then update its F value. If not, creates a new nav wrapper associated with this adjacent node and

	adds it to the open bins. (after calculating its f value).
void ResetOpenList()	Clears the open bin lists
void ResetNavWrapperDictionary()	Places all the Nav wrappers in the free pool of nav wrappers for use by another AStarSearch object. Then clears the nav wrapper map
void Reset()	Reset to starting conditions
void UnpackPath(List<Vector3Int> refPath)	Goes through the traversed nodes and generates a list of path points
bool CreatePath(NavNode startNode, NavNode endNode)	Begins processing a path from one start node to an end node
bool StartPath(Vector3Int startPos, Vector3Int endPos, List<Vector3Int> refPath, bool searchUsesDiagonals, int amortizeoverride = 0)	Takes the inputted coordinates and finds the nodes associated with them. Then begins to create a path.
void Process(int maxAmortize)	Run every frame. Only performs a certain number of iterations per frame based on the passed or overridden max Amortize value.

SimplePathFinding2D

Description:

A component that must be used in conjunction with a *Grid* component on a *GameObject*. Handles the creation of a navigation grid as well as processing any paths on this *Grid*.

Properties:

grid	A reference to the Grid component attached to the same GameObject as this component
navNodes	An array of navigation nodes/cells that make up the navigation grid. The 2D index of this array is referred to as Nav coordinates.
debugMarker	A optional GameObject that will be instantiated to display paths when this debug option is selected.
navMinX, navMinY, navMaxX, navMaxY	These are the max and min world coordinates of the nav grid.
initflag	A flag signifying this object has finished initialising
pathqueries	A list of path queries that are waiting or are still being processed
debugPathFindPoints,	A list of debug marker game objects used to display paths

debugDrawPathsFlag, debugDrawNavGridFlag	Flags used to display the marker paths or to display the tiles of the navigation tile map at run time
SelectedNavigationTileMap	Appears in the inspector. The tile map used as a mask to generate the navigation grid. 1 Tile represents 1 Navigation node in the grid.
BlockTile	Appears in the inspector. A specific Tile which represents that the Navigation node is blocked at this position
ColliderTileMaps	Appears in the inspector. A list of Tilemaps that contain collision data about your grid
NavigationMovementPenaltyTiles	Appears in the inspector. A list of different tiles (which should be part of the selected navigation tilemap), that can have different movement penalty values.
CorrespondingMovementPenalty	Appears in the inspector. A list that directly corresponds to the above list that specifies the movement penalty for each tile in that list
NavigationMapWidth, NavigationMapHeight	Appears in the inspector. User inputted width and height of the navigation grid with its centre at 0, 0, 0
DynamicAmortisation	Appears in the inspector. Linearly scales between min and max iterations per frame depending on the number of paths concurrently being processed. When this is turned off, defaults to the Maximum iterations per frame value
PathLoad	Appears in the inspector. The number of paths to process per frame. You may want to process a small number of paths per frame with large iterations or vice versa. Adjust this to get the performance you want
MinimumPathIterationsPerFrame, MaximumPathIterationsPerFrame	Appears in the inspector. The amount of iterations the A* algorithm will perform per frame for each path. Linearly scales between min and max if dynamic amortisation is turned on
DebugDrawPaths, DebugDrawNavGrid	Appears in the inspector. Flags that switch on debug functionality

DebugMarkerObject	Appears in the inspector. A pointer to a GameObject used for debug paths

Methods:

void DebugAddPathMarker(List<Vector3Int> refList)	Instantiates the DebugMarkerObject at every point in the passed in list
void DebugClearPathMarker()	Destroys the GameObjects instantiated when drawing a debug path
GridLayout.CellLayout GetGridLayout()	Returns the layout of the current grid (Rectangle, Hexagon, Isometric, IsometricZAsY
Vector3Int WorldToNav(Vector3 worldPos)	Transforms world coordinates to Navigation grid coordinates that can be used to index the navigation grid array
Vector3 NavToWorld(Vector3Int navPos)	Transforms navigation grid coordinates to world coordinates. Does some transformation based on the swizzle of the grid. Returns the centre point of the cell based on the layout
Vector3Int NavToTile(Vector3Int navPos)	Transforms navigation grid coordinates to tile coordinates
Vector3Int TileToNav(Vector3Int tilePos)	Transforms tile coordinate to navigation grid coordinates
NavNode GetNode(Vector3Int index)	Does a range check and returns the Nav node at the specified navigation coordinate/index
NavNode GetNode(int x, int y)	Does a range check and returns the Nav node at the specified navigation coordinate/index
NavNode GetNode(Vector3 pos)	Returns a nav node at the specified world coordinates
NavNode[,] GetNavGrid()	Returns the navigation grid array
void SetNavTileBlocked(Vector3Int tilepos, bool isBlocked)	Sets a tile on the navigation tilemap to be blocked
void SetNavTileMovementPenalty(Vector3Int tilepos, int val, TileBase tile = null)	Sets a tile on the navigation tilemap to have a movement penalty. You will need to specify which tile this is via the tile argument
bool GetNavNodesInBounds(Bounds bound, ref Vector3Int min, ref Vector3Int max)	Returns a rectangle with min and max corners that represents a rectangle of the nav nodes bounded by this Bounds object. Returns false if this fails, true if it succeeds.

void CreateNavMesh()	Creates a 2D array of NavNodes between the user specified width and height around the point 0,0,0. Uses the user selected tile map as a mask to create blocked and unblocked node as well as giving certain nodes movement penalties. If debug mode is not selected all tiles from the navigation tile map will be removed.
LinkedList<AStarSearch> GetPathQueries()	Returns the list of all the current path queries
bool IsInitialised()	Returns a flag that signifies whether this component has been initialised or not
void Start()	Called on component start up
void Update()	Update is called once per frame