Please visit https://anypath.bartvandesande.nl for the complete documentation.

Welcome to AnyPath

AnyPath is a completely generic, lightning fast A* pathfinding solution for Unity. Create your own data structure from scratch or use any of the included graphs as a starting point.

AnyPath uses Unity's job system and the Burst compiler. All of the heavy lifting is done on multiple threads and by highly optimized burst compiled code. Because all of the customizability is done via generic type parameters, all of the code is generated at compile time. This means there is zero overhead in defining custom functionality. On top of that, there's an efficient managed layer that hides the complexity of managing the native jobs, making it extremely easy to use.

Click here for a basic walkthrough or check out the API reference for a complete overview.

Features

Version 1.2 Updates:

- LineGraph added. A versatile 3D waypoint graph with support for queries from and to anywhere along the graph lines/edges
- NodeGraph added. A simple node to node 3D graph.
- Overall pathfinding performance boosted by using ref comparisons where possible
- New octree and quadtree implementation (https://github.com/bartofzo/NativeTrees), significantly boosting raycast performance for the NavMesh and PlatformerGraph
- Closest location query for PlatformerGraph, useful in a top down context
- Internal trees of NavMesh and PlatformerGraph are public now, useful for advanced location queries

Version 1.1 Updates:

- NavMeshGraph: Fast welding of vertices together using Unity's Job system, useful for large graphs and frequent updates
- NavMeshGraph: Can be populated and calculated inside of a burst compiled job now, useful for frequent updates
- NavMeshGraph: AABB triangle overlap queries, useful for cutting out triangles in a given area
- Breaking change: The NavMeshGraph constructor using NativeArrays now makes an internal copy of the data. Which prevents memory safety issues and allows the graph to be generated on another thread.
- PlatformerGraph: Fast welding of vertices together using Unity's Job system, useful for frequent updates
- PlatformerGraph: Can be populated and calculated inside of a burst compiled job now, useful for frequent updates
- PlatformerGraph: Graph can be drawn using the PlatformerGraphDrawer with automatic joining of vertices, making it extremely easy to generate a graph on the fly
- Fast copying of NavMesh, Platformer Graph, Quad- and Octree
- Tested with Unity 2021.3

Version 1.0

- Completely customizable graph data structures using generic types
- Grids or any number of dimensions are supported
- Leverages Unity's Job system and the Burst compiler, so performance and multithreading by default
- Works in jobs and with Unity ECS by utilizing the core building blocks in the AnyPath.Native namespace.
- A managed layer on top of the native code that is extremely easy to use
- Fully automated memory management (Non-ECS only). No need to worry about multi threading or race conditions, even when rebuilding your graph structure live.
- "Evaluate Only" requests that skip reconstructing a path in cases where it is sufficient to only check if a path exists, saving resources!
- Native support for adding extra stops in between the start and the goal of a request.
- Planning by giving a request a set of target objects

- Target picking by evaluating a set of target objects. Automatically generating a path to it.
- Target picking by evaluating which target is closest on the graph.
- Validating and reserving targets (Non ECS) without race conditions.
- "Edge Modifiers" that allow for applying small changes to a graph without the need to rebuild the entire graph. A door being open or closed for instance.
- Batch pathfinding operations. Evaluate multiple paths in a single job.
- ALT heuristics for any type of graph, which can significantly speed up pathfinding on very large and complex graphs.
- Object pooling is used where possible, so very low GC pressure
- 2D square grid
- Hexagonal grid
- 3D navigation mesh with support for curved surfaces
- A specialized platformer 'waypoint' graph
- Full source code included

The built in graphs also come with some utility classes that are neccessary for them to work properly, which you can use for your own custom graphs or any other kind of use you see fit:

- Burst compatible quadtree and octree implementations optimized for raycast queries.
- A 3D mesh welder utility, which welds together very close vertices on meshes.
- Burst compatible priority queue
- String pulling algorithms for straightening paths on a navigation mesh
- String pulling algorithm for usage with realtime steering behaviour
- Various geometry and math related utility functions

Who's this for?

AnyPath was designed for intermediate to advanced programmers with customizability in mind. If you just need a drop in solution for AI, then this framework is not for you. If you need total control over your data layout and fast pathfinding, then AnyPath is for you.

What's not included?

- No agent code, you write the movement code yourself. AnyPath only provides a (processed) path to follow
- No dynamic obstacle avoidance, as this is beyond the scope of A* and very game dependant
- Graph serialization (except for the platformer graph), as usually the graph can be easiliy generated at runtime. You can write your own serialization though.
- The platformer graph is the only graph which has a built in edit tool, other data structures need to be generated at runtime.
- No bi-directional A* search, search is always from start to goal

Requirements

- Unity 2020.3 or higher
- Burst package
- Unity.Collections package Look at the installation instructions here.

Contact

If you have any questions or encounter any bugs, feel free to contact me at anypath@bartvandesande.nl.

Basic Usage

Installation

Installation requirements:

- Unity 2020.3 or higher
- Burst minimum version 1.4.11
- Unity Collections minimum version 0.15.0 If your project uses the Entities package, Unity Collections will already be included

Import AnyPath.unitypackage into your project. If you encounter any compilation errors, you may need to install these packages manually.

- Click on Window -> Package Manager
- On the packages dropdown, select Packages: Unity Registry
- Locate Burst and install the latest version
- Click on the + > Add package from GIT url
- Enter com.unity.collections and hit ENTER

Finding a path

To find a path using AnyPath, first create a PathFinder for the type of graph you want to search. Goto Window->AnyPath Code Generator to create a definition for the finder you want to use. Paste the generated code somewhere in your project.

We'll use the built in 2D grid as an example:

```
IEnumerator LetsFindAPath()
{
    // Create our Path Finder
    var finder = new SquareGridPathFinder();

    // We can build our request by chaining methods
    finder
        .SetGraph(myGrid)
        .SetStartAndGoal(new SquareGridNode(0, 0), new SquareGridNode(10, 10));

    // Make the request and wait for the result
    yield return finder.Schedule();

    if (finder.Result.HasPath)
    {
        // Print out our path
        foreach (var edge in finder.Result.Edges)
        {
            Debug.Log(edge);
        }
    }
    else
    {
        Debug.Log("No path was found!");
    }
}
```

Evaluating a path

if you just need to know if there is a possible path, we can use an PathEvaluator. Additionaly, we can specify multiple stops in between.

```
IEnumerator OnlyEvaluateAPath()
{
    // Create our Path Finder
    var finder = new SquareGridPathEvaluator();

    // We can also just use properties
    finder.Graph = myGrid;

    // the first stop acts as the starting position
    finder.Stops.Add(new SquareGridNode(0, 0));

    // we'd like to make an in-between stop at 5,5
    finder.Stops.Add(new SquareGridNode(5, 5));

    // and finally arrive at 10, 10
    finder.Stops.Add(new SquareGridNode(10, 10));

    // Make the request and wait for the result
    yield return finder.Schedule();

    Debug.Log("Path found: " + finder.Result.HasPath);
}
```

Creating a Graph

Because AnyPath uses the Burst compiler, all graphs must be defined as structs. Where and how your store your graphs is completely up to you.

```
public class HexGridContainer : MonoBehaviour
{
    // assigned via inspector
    public List<(int2, float)> cells;

    // agents would reference the container and access this field
    public HexGrid Grid { get; private set; }

    private void Start()
    {
        // The way you create a graph is completely dependant on your own definition.
        // The only restriction is it has to implement IGraph<,> and must be a struct, since the requests will be burst compiled.
        Grid = new HexGrid(new int2(0, 0), new int2(1000, 1000), HexGridType.UnityTilemap, cells, Allocator.Persistent);
    }

    private void OnDestroy()
    {
        // make sure to dispose the graph when the container destroys.
        // this automatically takes care of any possible in flight requests.
        Grid.DisposeGraph();
    }
}
```

Disposing Graphs

Since the graphs use NativeContainers internally, they need to be disposed. Luckily, AnyPath provides the extension method DisposeGraph which can be called when the graph needs to be disposed. DisposeGraph delays the disposal until all requests that are currently operating on the graph are finished. As a result, you can call DisposeGraph on a graph that will be replaced and you don't have to worry about any possible in flight pathfinding requests that were still running on it.

```
// Requests all use this grid
public static HexGrid Grid { get; private set; }

public void BuildNewGraph()
{
    // Actual disposal is postponed until the requests that use the old graph are finished
    Grid.DisposeGraph();

    // We can safely replace the grid with a newer version which new requests will use
    Grid = new HexGrid(new int2(0, 0), new int2(1000, 1000),
        HexGridType.UnityTilemap,
        GetMostRecentCells(),
        Allocator.Persistent);
}
```

Graph Types

AnyPath comes with a built in set of graphs that can be used directly, or serve as a starting point for customizing to your specific needs.

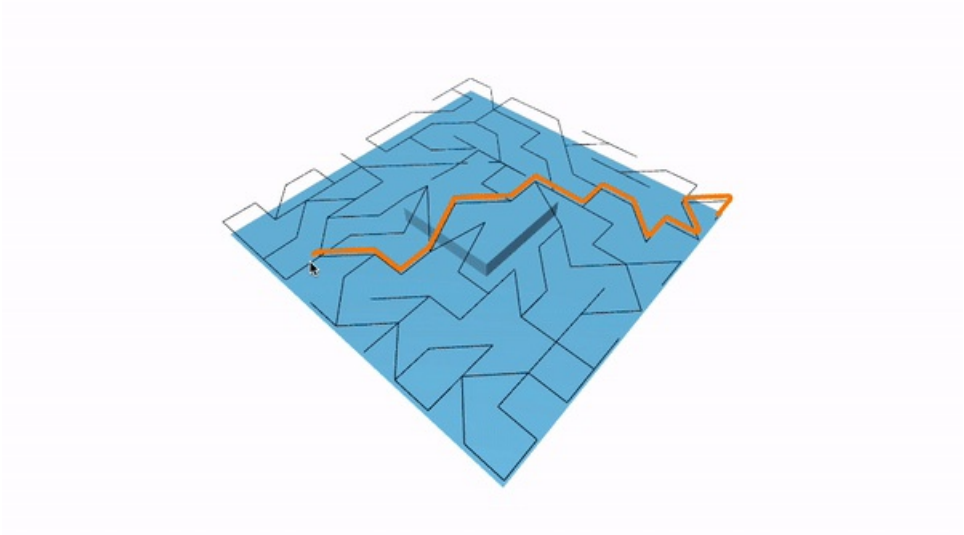
Navigation Mesh

Navigation mesh with the following features:

- Arbitrarily curved worlds.
- Path straightening (string pulling) which also works on curved worlds.
- Realtime steering target positions with string pulling, which can be useful in conjunction with obstacle avoidance (which is not included)
- Constructed from a set of vertices and triangles, so compatible with Unity's NavMesh generation, your own procedure or any other format (.obj, .fbx etc...)
- Smart navigation to exact positions within triangles, produces believable paths even with messy triangulation
- Per triangle cost
- Per triangle flags for excluding certain areas from a query
- Optimized raycast queries on the mesh, useful for knowing where the agent's location on the mesh is or collision detection.
- Also suitable for 2D navigation meshes in any plane (XY, XZ) or rotation for that matter.
- New (v1.1): Fast welding of vertices together using Unity's Job system, useful for frequent updates
- New (v1.1): Graph can be pre-allocated and then populated and calculated from within a burst compiled job, useful for frequent updates
- New (v1.1): AABB triangle overlap queries, useful for cutting out triangles in a given area

See AnyPath.Graphs.NavMesh form more information.

Line Graph



A graph that describes 3D edges/lines between points. Pathfinding can be done from any location on these edges to any location on another edge. Making it more intuitive foragents that can move smoothly along these edges.

Very similar to AnyPath.Graphs.PlatformerGraph, but in 3D.

Features:

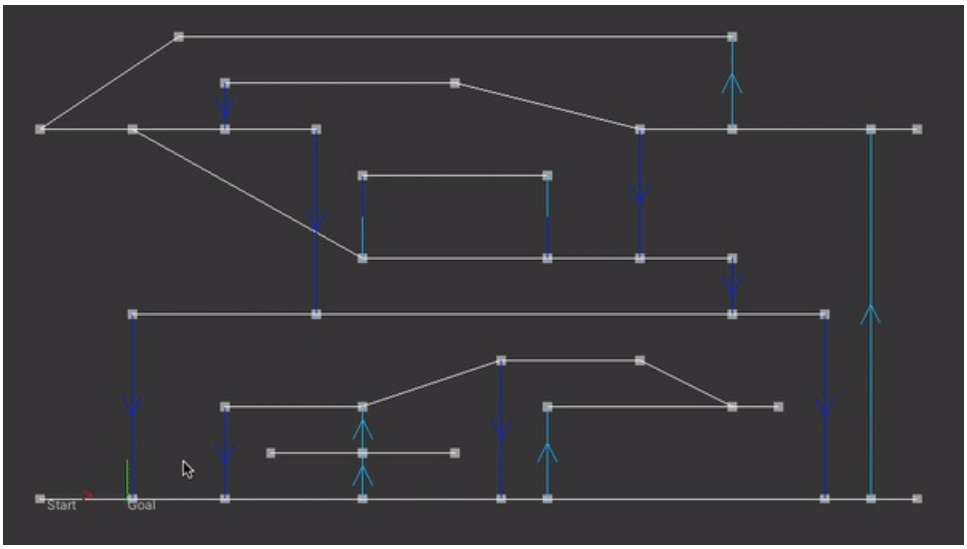
- Large worlds don't require a massive amount of nodes for every position an agent can be at.
- Paths are calculated from edge to edge, which gives far better results when an agent is not located near a corner node.
- Edges can be tought of as lines/roads.
- Edges themselves can have additional properties.
- An agent can have any floating point position on an edge and can navigate to any position on another edge. So your pathfinding query doesn't have to start and end at the exact node positions but can be anywhere in between.

Additional features:

- Nearest edge/road queries
- The resulting path is a list of segments containing all the information, the exact enter and exit positions on each edge and it's flags, so once you have a path, there doesn't have to be any need to do additional raycasting or collision detection (with the static world at least).
- Per edge cost
- Directed and undirected edges
- Per edge flags for excluding certain areas
- AnyPath.Graphs.Line.SceneGraph, a Platformer Graph Scene Editor tool to design a platformer graph inside of your scene.

See AnyPath.Graphs.Line for more information.

Platformer Graph



A graph specifically designed for 2D platformer types of games, but can also be used as a top down 'roads' system.

What makes this graph unique is that the edges themselves play the main part, not the nodes they connect. This has several advantages:

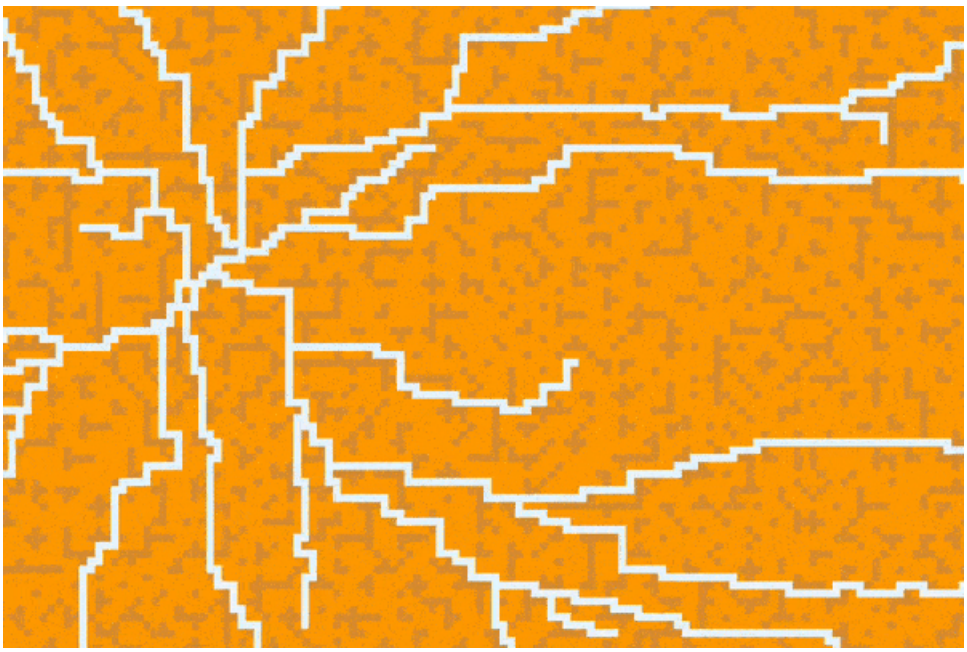
- Large worlds don't require a massive amount of nodes for every position an agent can be at.
- Paths are calculated from edge to edge, which gives far better results when an agent is not located near a corner node.
- Edges can be tought of as surfaces where an agent can walk for platformer type games, or as 'roads' for top down games.
- Edges themselves can have additional properties. For example, there can be a directed jump edge between nodes with a higher cost associated with them, connecting two surface edges. Another example could be a fall edge, or fly edges.
- An agent can have any floating point position on an edge and can navigate to any position on another edge. So your pathfinding query doesn't have to start and end at the exact node positions but can be anywhere in between.

Additional features:

- Optimized raycast queries for knowing at which edge/road the agent currently is located. For example, raycasting down to find the surface the agent walks on.
- Nearest edge/road queries
- The resulting path is a list of segments containing all the information, the exact enter and exit positions on each edge and it's flags, so once you have a path, there doesn't have to be any need to do additional raycasting or collision detection (with the static world at least).
- Per edge cost
- Directed and undirected edges
- Per edge flags for excluding certain areas
- New (v1.2): Id's per edge, allowing for easy mapping back to MonoBehaviour script via GetInstanceId.
- New (v1.2): Closest edge query now accepts a delegate to check for obstructions in the line of sight.
- New (v1.1): Graph can be pre-allocated and populated from within a burst compiled job. Useful if you need fast, frequent updates.
- New (v1.1): Fast welding of vertices together using Unity's Job system, useful for frequent updates
- New (v1.1): Graph can be drawn using the PlatformerGraphDrawer with automatic joining of vertices, making it extremely easy to generate a graph on the fly
- AnyPath.Graphs.PlatformerGraph.SceneGraph, a Platformer Graph Scene Editor tool to design a platformer graph inside of your scene.

See AnyPath.Graphs.PlatformerGraph for more information.

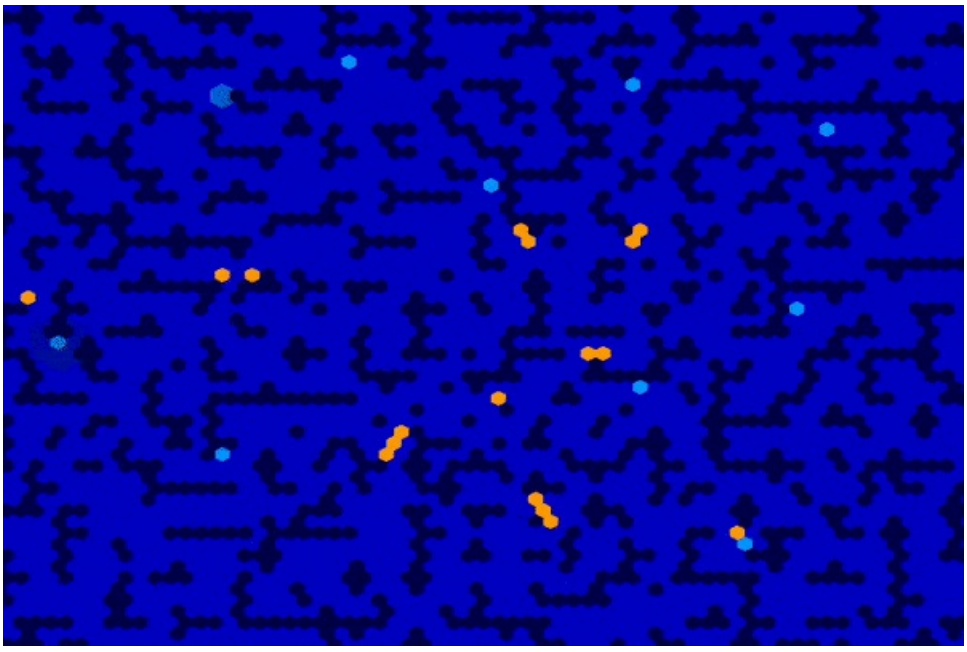
Square Grid



A simple square grid structure with support for per tile cost and four or eight neighbours per tile.

See AnyPath.Graphs.SquareGrid for more information.

Hexagonal Grid



A simple hexagonal grid structure with support for per tile cost.

See AnyPath.Graphs.HexGrid for more information.

Node Graph

NodeGraph

A simple graph structure that connects vertices/nodes together in 3D space. Nodes can have additional cost and flags associated with them. This is a good option if you only have fixed nodes in space and don't require agents to be able to find paths "in between" nodes. Agents should also keep track of the node they are currently at themselves.

The <xref:AnyPath.Graphs.LineGraph> is a more advanced representation of a 3D graph.

See AnyPath.Graphs.Node for more information.

Finders

The Finder class and it's derivations manage creating and scheduling a path finding request and hold the result when the request is finished. Similar to how a UnityWebRequest is made.

Behind the scenes, the request is passed on to Unity's Job system, which runs it on a different thread and with the massive speedup the Burst compiler gives, so your game code can keep on running. Once your request is in flight, the finder object may not be modified until the result is back.

Awaiting a request can be done with a callback, or with a yield return statement inside of a coroutine. When the job has completed, all of the data is converted back into the managed domain, so you don't have to worry about the memory management that the Job system imposes.

The base PathFinder class has the following definition:

```
public class PathFinder<TGraph, TNode, TSeg, TProc>
```

You'll notice there are 4 (!) different generic type parameters:

- TGraph: The type of graph the finder operates on
- TNode: The type of location/node that the graph contains
- TSeg: The type of path segments the path will be made up of
- TProc: The type of (post) processing to apply to the path

Luckily, you don't need to fill these in yourself. AnyPath's built in graphs all come with predefined finders that fill in these parameters. If you've created your custom graph type, AnyPath comes with a tool to automatically generate the concrete types suitable for your graph. See Custom Graphs for more information.

Building the request

Setting data on a finder can be done directly via it's properties, or with extension methods allowing fluid style method chaining. See FinderExtensions in the API documentation for a detailed description on the extension methods.

Before making a request, a few properties need to be set:

- Graph: Set the graph the request should be evaluated on.
- Stops: The stops contain the start and goal location for the request. If you add more than two stops, AnyPath will try to find a path that visits all stops in the order in which they were added.

If you need more control, the Settings property can be set with a new <xref:AnyPath.Native.AStarSettings> value.

You can then either use the Run() method to directly run the request on the main thread, or the Schedule() method to use Unity's Job system to execute it on another thread. The Schedule method returns an IEnumerator which can be used to suspend a coroutine until the request is finished. Additionaly, you can bind to the Completed event to receive a callback.

After the request has completed, the Result property of the finder will contain the result of the request. Depending on the type of finder, this can be a Path<TSeg> object or an Eval struct.

> **⚠ WARNING**
>
> When a finder's request is in flight, you can not modify it's properties. This is to ensure the result that comes back always matches the request's settings. If you want to reuse the same finder for a new request, you should always call Clear(ClearFinderFlags) before making a new request.

Finder Types

Below is an overview of the different type of finders AnyPath has built in. Each of these finder types has a corresponding Evaluator, which does not reconstruct a path and is cheaper to use when all you need to know is if a path exists.

PathFinder

Just finds a path from a start to a goal, with the possibility of adding stops in between.

MultiPathFinder

A batch operation that finds multiple paths in one job.

OptionFinder

The option finder can assist in planning which target to choose from a set of options. Each option can be defined as a start and goal location with the possibility of adding stops in between. An option is associated with a user defined object of any type.

When the request is made, the options are evaluated in the order in which they were added. Then on the first option that results in a valid path, the evaluation stops and that option is returned.

PriorityOptionFinder

Similar to the OptionFinder, but prioritizes options using an IComparer that must be provided. This can be done with the SetComparer extension method or by directly assigning the Comparer property with your custom IComparer.

CheapestOptionFinder

Similar to the OptionFinder but results in the option that is the cheapest to reach from the starting location. The process is optimized in such a way that options that have a heuristic value larger than the currently found cheapest option are discarded right away.

Validating and Reserving

Consider the situation where multiple agents are given the same set of target objects to visit but you need every agent to pick a unique target. Every OptionFinder can be extended with a custom validation and reserval function on your target objects, which can help to ensure no agents will plan a path to the same object. This is shown in the included HexGrid example.

```
public class HexGridGoalValidator : IOptionValidator<HexGridGoal>, IOptionReserver<HexGridGoal>
{
   public static HexGridGoalValidator Instance { get; } = new HexGridGoalValidator();

   // A seeker may only consider goals that aren't already seeked
   // This function is called before any attempt at finding a path to this target is made. When a path is found
   // this is called again to ensure no other seeker reserved the goal in the time it took to run the algorithm on another thread.
   // If the target was reserved in that time, the request is restarted and other options will be considered.
   public bool Validate(HexGridGoal target)
   {
      return !target.IsSeeked;
   }

   // Reserves a goal so that no other seekers may target it
   public void Reserve(HexGridGoal option)
   {
      option.IsSeeked = true;
   }
}
```

Options are validated twice. Once before the actual A* starts, and once when the result is back. The second validation occurs because the object may have been invalidated in the time it took for the request to complete. If the validator returns false on the second call, the option

is discarded and the request is retried without the options that were already marked as invalid.

> **NOTE**
>
> Be careful with clearing an option finder that's in flight in conjunction with a reserver. This may leave a dangling reservation that will never get cleared, depending on your implementation.

> **NOTE**
>
> Altough rarely the case, you can use the MaxRetries property to specify how many retries may be attempted. Keep in mind that a retry will only occur if the validation fails on the second call (which has a small chance since the change must have happened within a few ms), also, there must still be any valid options left for a retry to occur.

DebugFinder

Finally, there exists the debug finder. The debug finder returns the path in original node format, so you can debug it against what your path processor is doing. Also, an array containing all of the nodes A* visited, which can be helpful in determining the quality of your heuristic function, since for optimal performance, this array should be as small as possible.

In depth explanation

AnyPath was designed to be as modular as possible. Because all of the heavy lifting is done in Burst compiled jobs, there are a few limitations to what subset of C# AnyPath can use. One of these is that we can not use classes, so we can't use inheritance. The way AnyPath circumvents this limitation while still remaining modular is by using structs that implement interfaces. Every component is passed in as a generic type parameter and as such, the burst compiler can generate very fast code as if the code contained in these structs was there in the function itself.

This does result in a lot of generic type parameters however, luckily, there is the Code Generation utility to assist with defining which components you need for your project.

If you don't use ECS, the AnyPath.Managed finders combined with code generation hide all this complexity for you.

Here's an overview of the components that make up AnyPath:

Graph <TGraph>

Defines a graph data structure that can be used to perform pathfinding queries on.

Nodes <TNode>

The nodes that make up the graph. The only constraint is that a node has to implement IEquatable

Heuristic Provider <TH>

Allows for plugging in a function that provides a cost estimate between two nodes.

Edge Modifier <TMod>

Allows for plugging in a function that alters the cost for an edge between two nodes, without requiring to update the entire graph.

Path Processor <TProc>

After the A* algorithm has found a sequence of nodes that make up the path, it is fed to the path processor. Here, additional processing can be performed on the path. An example of this would be converting a sequence of triangles into a list of corner points.

Segment <TSeg>

Specifies the output type for final path. In simple cases, this can be the same type as the nodes if no path processing is performed. But this can be of any type, as long as the path processer provides the conversion from the list of nodes to the list of path segments.

Performance considerations and heuristics

The performance of A* depends mostly on how good we can estimate the actual cost of travelling from A to B. The better this estimate actually matches the true distance from A to B, the less nodes A* has to look at before finding a path.

A heuristic function is said to be admissable as long as it not overestimates the cost between two nodes. As long as your heuristic function is admissable, A* is guarantueed to always return the shortest path possible between two nodes.

One way to lower the amount of nodes that A* has to expand into, is to slightly over estimate the cost. The caveat being that the resultign path may not be the actual shortest path, but in many cases, this will be good enough.

Here's a way to implement a heuristic provider for a 2D grid that overestimates the cost:

```
public struct RelaxedGridHeuristics : IHeuristicProvider<int2>
{
    public readonly float relaxation;
    private int2 goal;

    public RelaxedGridHeuristics(float relaxtion)
    {
        this.relaxation = relaxtion;
    }

    public void SetGoal(int2 goal)
    {
        // this will be called before A* starts, all measurements happen from x to the goal.
        // store the goal value internally for later usage
        this.goal = goal;
    }

    public float Heuristic(int2 x)
    {
        // the higher the value of relaxation is, the more A* will try to expand in a straight line towards the destination.
        // this can greatly reduce the amount of cells that are expanded into, but may not always result in the truly shortest path
        // in many cases however, this is not a big deal, especially if the over estimation is low.
        // the dot(abs(x - goal), 1f) calculation is just a fast way to calculate manhattan distance on a grid.
        return (1 + relaxation) * dot(abs(x - goal), 1f);
    }
}
```

ALT

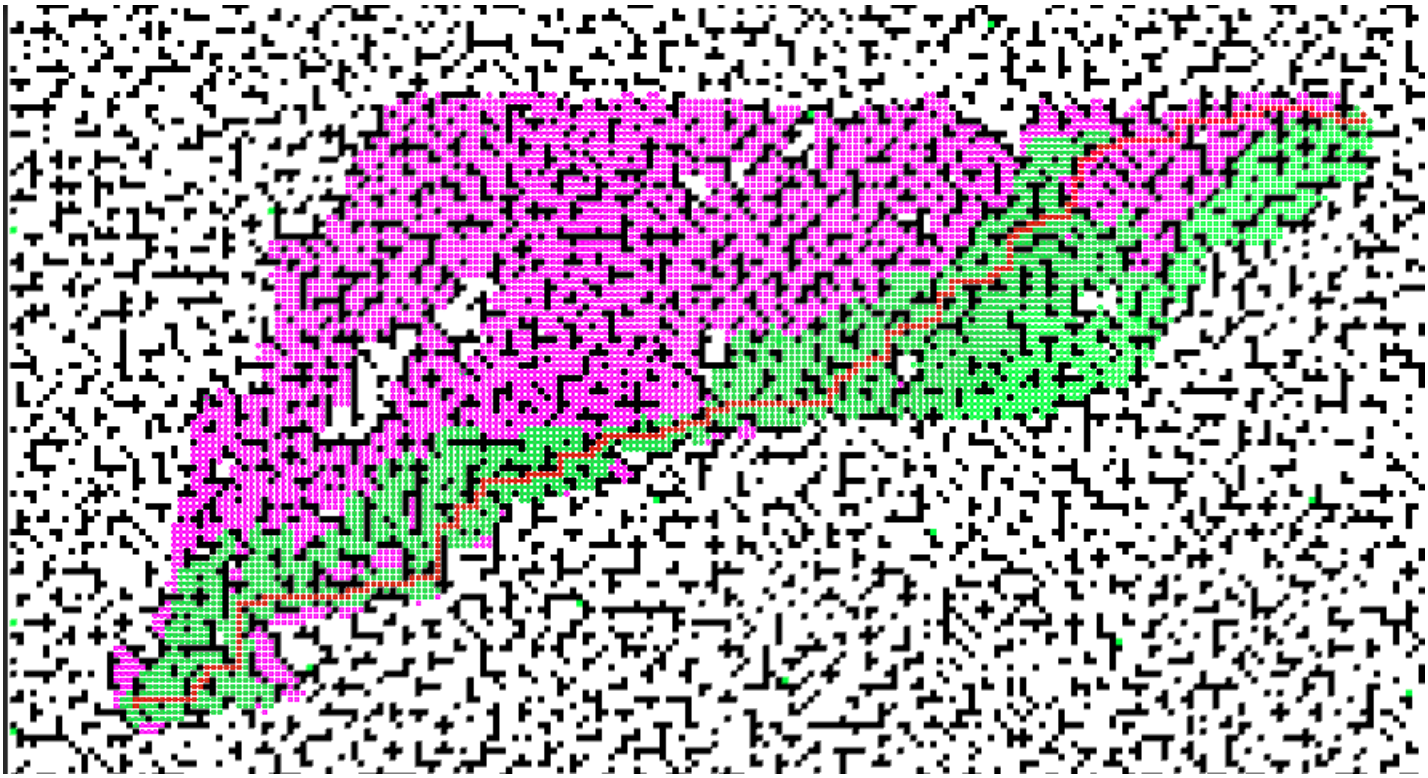
We can do much better however, by using ALT heuristics.

ALT stands for Admissable Landmarks and Triangle Inequality and is explained in this paper: https://www.microsoft.com/en-us/research/publication/computing-the-shortest-path-a-search-meets-graph-theory/?from=http%3A%2F%2Fresearch.microsoft.com%2Fpubs%2F154937%2Fsoda05.pdf

The great thing about ALT heuristics is that it works on any type of graph. So this heuristic provider can be used with all of the built in graphs but also on a custom graph type without requiring extra coding.

The trade off for using ALT heuristics is pre processing time and memory consumption. ALT works by selecting a small amount of nodes as "Landmarks". These landmarks are spread out over your graph, ideally placed "behind" frequent starting and goal locations. In the preprocessing stage, distances to and from every landmark are computed. When running A*, these landmark distances can then be used to produce a far better estimate of the true distance between two nodes. Especially on true graph like structures (where eucludian distance performs pretty bad) and maze like structures.

The image below shows the difference between ALT and regular heuristics. The red squares represent the path, the purple squares represent the cells that A* expanded into without using ALT heuristics, the green ones with ALT.



Generating ALT heuristics for your graph

There are 2 stages to generating ALT heuristics for your graph. The first is, selecting which landmarks to use. AnyPath provides two ways to automatically do this. Alternatively, you can manually specify the locations for your landmarks if that works better for your use case.

Once you have your landmark locations, we can contruct the ALT heuristic provider.

```
// This will store our landmarks, a maximum of 31 is supported
SquareGridCell[] landmarks = new SquareGridCell[31];

// Scans the entire grid and selects landmarks that are spaced apart evenly
LandmarkSelection.SelectFarthestLandmarksUndirected(ref grid, grid.GetEnumerator(), landmarks);

// Create our heuristic provider
altHeuristics = new ALT<SquareGridCell>(Allocator.Persistent);

// Compute it for the landmarks we've selected
altHeuristics.ComputeUndirected(ref grid, landmarks);
```

Now for our pathfinding queries, we generate the code for a Finder that uses this provider. We then only need to assign the heuristic provider before making the query:

```
pathFinder
   .SetGraph(grid)
   .SetHeuristicProvider(altHeuristics) // assign our ALT heuristics
   .SetStartAndGoal(start, goal)
   .Run(); // runs on the main thread immediately
```

NOTE

Keep in mind that the ALT heuristic provider contains NativeContainers and needs to be disposed when it is no longer used.

Directed and undirected

Preprocessing for ALT heuristics is not a cheap operation. The computation of landmark selection and preprocessing is burst compiled and multithreaded where possible. It can still take quite some time for large graphs however. If you know your graph is undirected (which most

are), we can cut the processing time in half by using the undirected counterparts of the methods. Of all of the built in graph types, only the PlatformerGraph supports directed edges. And if you don't use directed edges in the platformer graph, you can treat it as an undirected graph.

> **NOTE**
>
> To compute ALT heuristics for directed graphs, you need to compute a <xref:AnyPath.Native.ReversedGraph{TSeg}> first.

ALT and graph updates/edge mod

As long as edge costs increase, it is not strictly neccessary to recompute ALT heuristics. The bigger the change however, the more off ALT heuristics will be and you will lose some of the performance benefits.

Serializing ALT heuristics

If you have very large static worlds, it may be benificial to serialize the heuristic data and compress it. AnyPath provides the ReadFrom and WriteTo extension methods for usage with a BinaryReader/Writer. You'll only need to provide a function that writes and reads the node data:

```
public void Read(ALT<SquareGridCell> alt, BinaryReader reader)
{
   alt.ReadFrom(reader, reader =>
   {
      // read X and Y and convert back into cell struct
      return new SquareGridCell(new int2(reader.ReadInt32(), reader.ReadInt32()));
   });
}

public void Write(ALT<SquareGridCell> alt, BinaryWriter writer)
{
   alt.WriteTo(writer, (writer, cell) =>
   {
      // Just write the X and Y components
      writer.Write(cell.Position.x);
      writer.Write(cell.Position.y);
   });
}
```

> **NOTE**
>
> Benchmarking is the way to go here. Your results may vary depending on the size and complexity of your graph. On my machine, computing for a 1000x1000 sized grides takes about 20ms with burst compilation enabled. Serializing the data in such a case may not be useful.

Customization

The biggest power of AnyPath lies in the fact that you can easily extend it to use your own custom data structures.

To define your own graph type, create a struct that implements the IGraph interface. Then all that is needed is implementing the Collect function that let's the algorithm know which nodes connect to which nodes.

Additionaly, you can make an implementation of IHeuristicProvider to match your custom data structure.

The following example shows a very simple 2D grid that uses an array to store it's values:

```
namespace AnyPath.Examples
{
    /// <summary>
    /// An example of a simple 2D grid that uses a 1D array as backing
    /// </summary>
    public struct SimpleGrid : IGraph<int2>
    {
        private readonly int width;
        private readonly int height;
        private NativeArray<bool> cells;

        public SimpleGrid(int width, int height, Allocator allocator)
        {
            this.width = width;
            this.height = height;
            this.cells = new NativeArray<bool>(width * height, allocator);
        }

        /// <summary>
        /// Sets a wall at a given position. This cell will be unwalkable.
        /// </summary>
        public void SetWall(int2 position) => cells[PositionToIndex(position)] = true;

        /// <summary>
        /// Clears the wall at a given position. The cell will be walkable.
        /// </summary>
        public void ClearWall(int2 position) => cells[PositionToIndex(position)] = false;

        /// <summary>
        /// Returns wether there is a wall at a given position.
        /// </summary>
        public bool IsWall(int2 position) => cells[PositionToIndex(position)];

        /// <summary>
        /// Converts a position into a 1D index into the backing array
        /// </summary>
        private int PositionToIndex(int2 position) => position.x + position.y * width;

        /// <summary>
        /// Returns wether a position is in bounds of the grid
        /// </summary>
        public bool InBounds(int2 position) => position.x >= 0 && position.x < width && position.y >= 0 && position.y < height;

        private static readonly int2[] directions =
        {
            new int2(0, 1), new int2(1, 0), new int2(0, -1), new int2(-1, 0),
        };

        /// <summary>
        /// AnyPath will call this function many times during a pathfinding request. It should return the neighbouring cells from node
        /// that are walkable.
        /// </summary>
        public void Collect(int2 node, ref NativeList<Edge<int2>> edgeBuffer)
```

```
        {
            // Loop over the four directions
            for (int i = 0; i < directions.Length; i++)
            {
                var neighbour = node + directions[i];

                // Skip this node if it's out of bounds, or if it is a wall
                if (!InBounds(neighbour) || IsWall(neighbour))
                    continue;

                // This neighbour is a valid location to go to next, add it to the edge buffer with a cost of one
                edgeBuffer.Add(new Edge<int2>(neighbour, 1));
            }
        }

        // Dispose of the inner NativeArray:
        public void Dispose() => cells.Dispose();
        public JobHandle Dispose(JobHandle inputDeps) => cells.Dispose(inputDeps);
    }

    /// <summary>
    /// This struct will be used to provde a heuristic value for our pathfinding queries
    /// </summary>
    public struct SimpleGridHeuristc : IHeuristicProvider<int2>
    {
        public float Heuristic(int2 x, int2 goal)
        {
            // Manhattan distance
            return math.dot(math.abs(x - goal), 1f);
        }
    }
}
```

NOTE

Because the algorithm runs in burst accelerated code, only a limited subset of C# is allowed in Graph, Path processing and Edge modifier code. See the following: https://docs.unity3d.com/Packages/com.unity.burst@0.2-preview.20/manual/index.html#cnet-language-support

Path Post Processing

Certain graph types can benefit from post processing the list of segments that is produced after a path has been found. For example, the included Navigation Mesh tries to straighten the path it has been found by applying a string pulling algorithm. See the API reference IPathProcessor<TNode, TSeg> for more details.

NOTE

All of the code that runs in the processor is still performed on the burst accellerated job, ensuring maximum speed.

Modifying and discarding edges on the fly

Say you have a graph that describes your static world but there are some locations contained in it that can have various states, like a door. You can implement your own edge modifer to incorporate such changes without the need to completely rebuild the graph. Saving significant overhead for large worlds.

See IEdgeMod<TNode> or the included 2D grid example on how to incorporate this.

Another use case is filtering out unwanted areas using a flag bitmask. See FlagBitmask<TNode> for details.

Creating Finders

To use finders on this graph, you could fill in all of the generic type parameters to the ones defined in your graph, but this is not ideal. AnyPath comes with a tool to quickly generate readable finder definitions. To use it, go to Unity -> Window -> AnyPath Code Generator.

Clicking on the Scan Graph Definitions button will scan your project for graph types. Selecting a graph type will generate the code for all of the finder definitions, saving you the hassle of filling in all the type parameters by yourself.

> **NOTE**
>
> If you need a custom post processor associated with the finders, change the NoProcessing<,> into the type name of your processor struct.

Combinding several graphs

```
// This is an example of how you could combine several graphs as 'sections'
// where each section could be updated without affecting other sections
// you would keep the section graphs as separate containers somewhere else, and run the pathfinding query
// on this struct, which doesn't allocate the containers by itself
// the downside is, you'll need to hardcode the amount of sections contained because there is no way to have
// an array of NativeContainers in burst compiled code

public struct ComposedGraph<TGraph, TNode> : IGraph<TNode>
    where TGraph : struct, IGraph<TNode>
    where TNode : unmanaged, IEquatable<TNode>
{
    public TGraph section1;
    public TGraph section2;
    public TGraph section3;
    public TGraph section4;

    public void Collect(TNode node, ref NativeList<Edge<TNode>> edgeBuffer)
    {
        // let each section decide if they have edges coming from this node
        // e.g. if the current node was in section 1 and section 2 is connected to it, section 2 will add a node and so on
        section1.Collect(node, ref edgeBuffer);
        section2.Collect(node, ref edgeBuffer);
        section3.Collect(node, ref edgeBuffer);
        section4.Collect(node, ref edgeBuffer);
    }

    // we don't need any disposal here but rather the individual graph sections themselves should be disposed of
    public void Dispose()
    {
    }

    public JobHandle Dispose(JobHandle inputDeps) => inputDeps;
}
```

Low level usage (Jobs and ECS)

The managed finders are just a layer of abstraction on top of the AnyPath.Native functions. All of the methods in the AnyPath.Native namespace are Burst compatible.

To do a low level pathfinding request, create an instance of AStar<TNode> and call the FindPath, EvalPath or one of the more complex extension methods found on AStarStops, AStarOption or AStarCheapestOption.

See below for an example.

A note on ECS

Entities can store dynamic elements in a so called DynamicBuffer component. Unfortunately, as for now, Unity does not provide a way to completely generalize between a DynamicBuffer and NativeList.

In order to fully support all the functionality AnyPath provides, a choice had to be made. That is why all of the native AnyPath methods operate on NativeList and NativeSlice structs. If you need to store your path in a DynamicBuffer, you can provide a temporary or persistent backing memory NativeList and copy from that list after. Here's an example:

```
// This can run as part of a burst compiled job, or in an ECS system
public static bool FindPathAndStoreInDynamicBuffer(ref SquareGrid grid, SquareGridCell start, SquareGridCell goal, ref
DynamicBuffer<SquareGridCell> path)
{
    // these could also be persistent in the system or job to prevent reallocation. Note that simultaneous requests need their own instance.
    var aStar = new AStar<SquareGridCell>(Allocator.Temp);
    var tempPathBuffer = new NativeList<SquareGridCell>(128, Allocator.Temp);

    aStar.FindPath(ref grid, start, goal,

        // default heuristic for a 2D grid
        default(SquareGridHeuristicProvider),

        // we don't use any edge modifiers
        default(NoEdgeMod<SquareGridCell>),

        // no path processing (just the raw node output)
        default(NoProcessing<SquareGridCell>),

        // the path will be appended to this list
        tempPathBuffer);

    if (!result.evalResult.hasPath)
        return false;

    // if you don't use DynamicBuffer, the path will be in tempPathBuffer now
    // otherwise, copy the path to the DynamicBuffer. This uses a memcpy internally so it's very fast
    path.CopyFrom(tempPathBuffer);

    return true;
}
```

NOTE

If Unity ever introduces a safe way to generalize between NativeList and DynamicBuffer via INativeList (specifically: a way to create a NativeSlice from INativeList) AnyPath will get an update so that it can directly operate on DynamicBuffer.

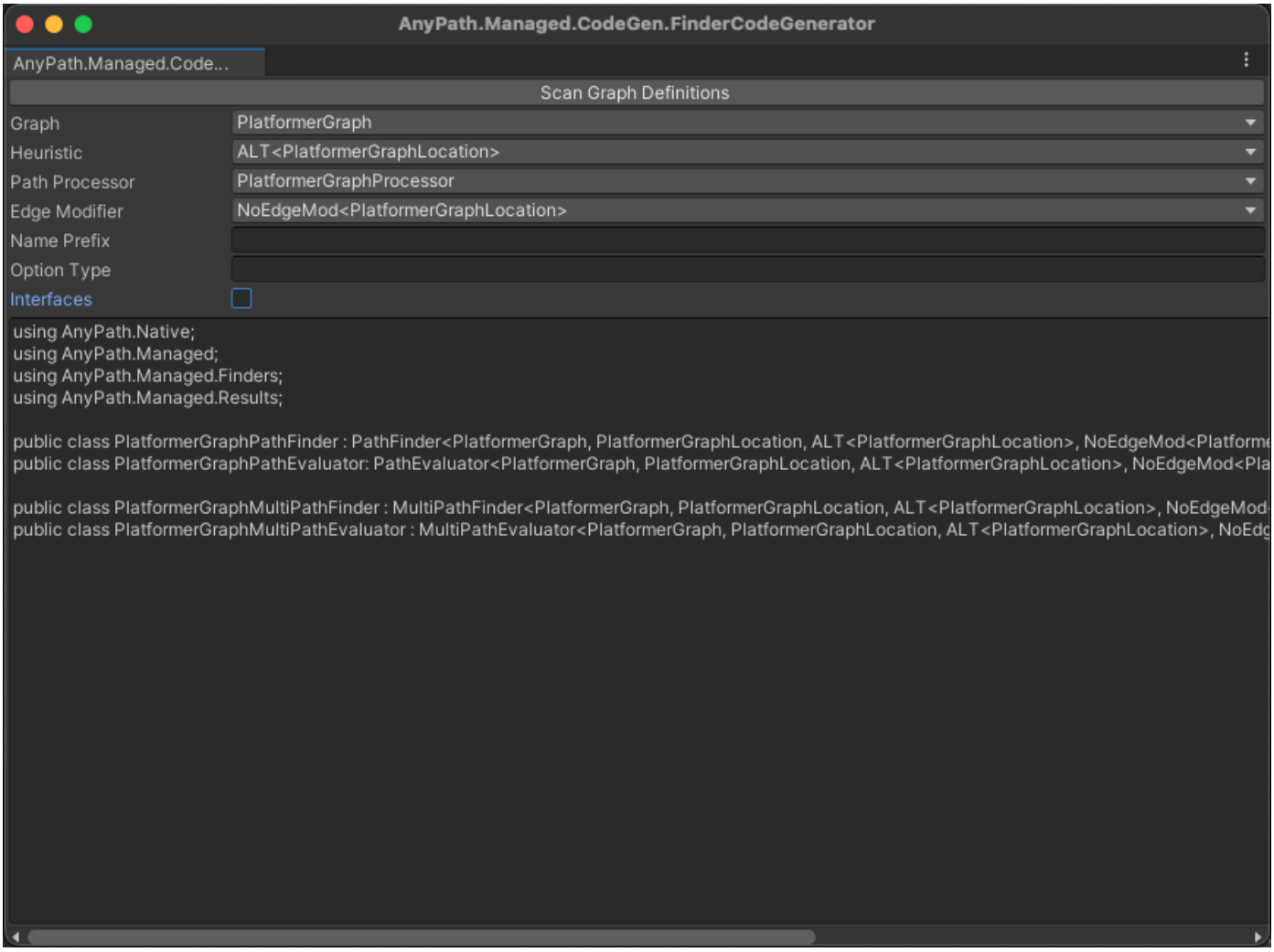
Alternative way

```
public static void AnotherWay(NavMeshGraph graph, NativeList<NavMeshGraphLocation> stops, ref NativeList<float3> path)
{
    // As an alternative to using the static AStar functions which require a lot of type parameters,
    // we can leverage the managed finder's Job struct to make our native query more readable.
    // we use the job struct and directly call Execute on it.
    // the NavMeshGraphPathFinder code can be generated using the code generator

    var job = new NavMeshGraphPathFinder.Job()
    {
        graph = graph,
        stops = stops,
        aStar = new AStar<NavMeshGraphLocation>(Allocator.Temp),
        path = path
    };

    job.Execute();
    job.aStar.Dispose();

    // path will now be filled with the corner points
}
```

Code Generation

AnyPath comes with a tool to quickly generate readable finder definitions. To use it, go to Unity -> Window -> AnyPath Code Generator.

Clicking on the Scan Graph Definitions button will scan your project for graph types. Selecting a graph type will generate the code for all of the finder definitions, saving you the hassle of filling in all the type parameters by yourself.



Graph

Select the graph type for which you want to generate finder class definitions for.

Heuristic

Select a heuristic provider.

Path Processor

Select a path processor to determine the output of a pathfinding query.

Edge Modifier

Select an edge modifier. Use NoEdgeMod for no modifier (default).

Name Prefix

Optionally provide a custom name prefix for the generated declarations.

Option Type

If you want to use any of the option finders, provide the type name of the object you're targeting. When filled in, additional option finder definitions will be generated.

Interfaces

Check to generate an interface which can provide interoperability between different finder types in your code.

There is an example for every built in graph type. These examples are heavily commented and should give you a good idea on how to use AnyPath.

SquareGrid

- Finds multiple paths in one job using the MultiFinder
- Demonstrates how to use an Edge Modifier
- Finds composed paths (multiple stops in between the start and goal)
- Demonstrates the usage of the DebugFinder
- Shows how to use ALT heuristics

HexGrid

- Uses OptionFinders (Cheapest and priority) with reservation and validation that can assist in AI planning.

NavMesh

- Shows how to use raycast queries against the mesh
- Shows how to straighten a path and get corner points to follow
- Also shows how realtime steering can be applied

PlatformerGraph

- Demonstrates the PlatformerGraph
- Shows how the graph can be edited in the Unity editor

Caveats

Here's an overview of some of the caveats.

Unreachable islands

Consider a scenario on a grid where a goal location is totally enclosed by walls. If the starting location is beyond those walls, A* will have to search the entire grid before knowing for certain that the goal is unreachable. There are several ways to mitigate this.

- The easiest method is to provide an upper bound on the maximum amount of nodes A* may expand into. This can be set with the MaxExpand property on a Finder.
- Do a simple Eval query from goal to start before attempting to find a full path from start to goal. This only works for undirected graphs.
- Use seperate graphs or flags to know beforehand if islands are connected

Standalone builds

Unity does not support open generic types for burst compiled jobs in generic jobs. If you use any of the OptionFinders, this means that the TOption type cannot be left open in the finder class declaration. This unfortuntately means you have to declare a finder class for every specific type you're targeting with an option finder. The code generater enforces this rule for you however.

Note that the jobs will work in the Unity editor, but an exception will be thrown in a standalone build if the job could not be burst compiled.