MERCUNA

Mercuna Ground Navigation
Unreal Engine User Guide

v2.7

MERCUNA

Contents

MERCUNA

Installing

The Mercuna middleware is integrated into Unreal Engine as a standard plugin compatible with **Unreal Engine 5.0.1 or later**.

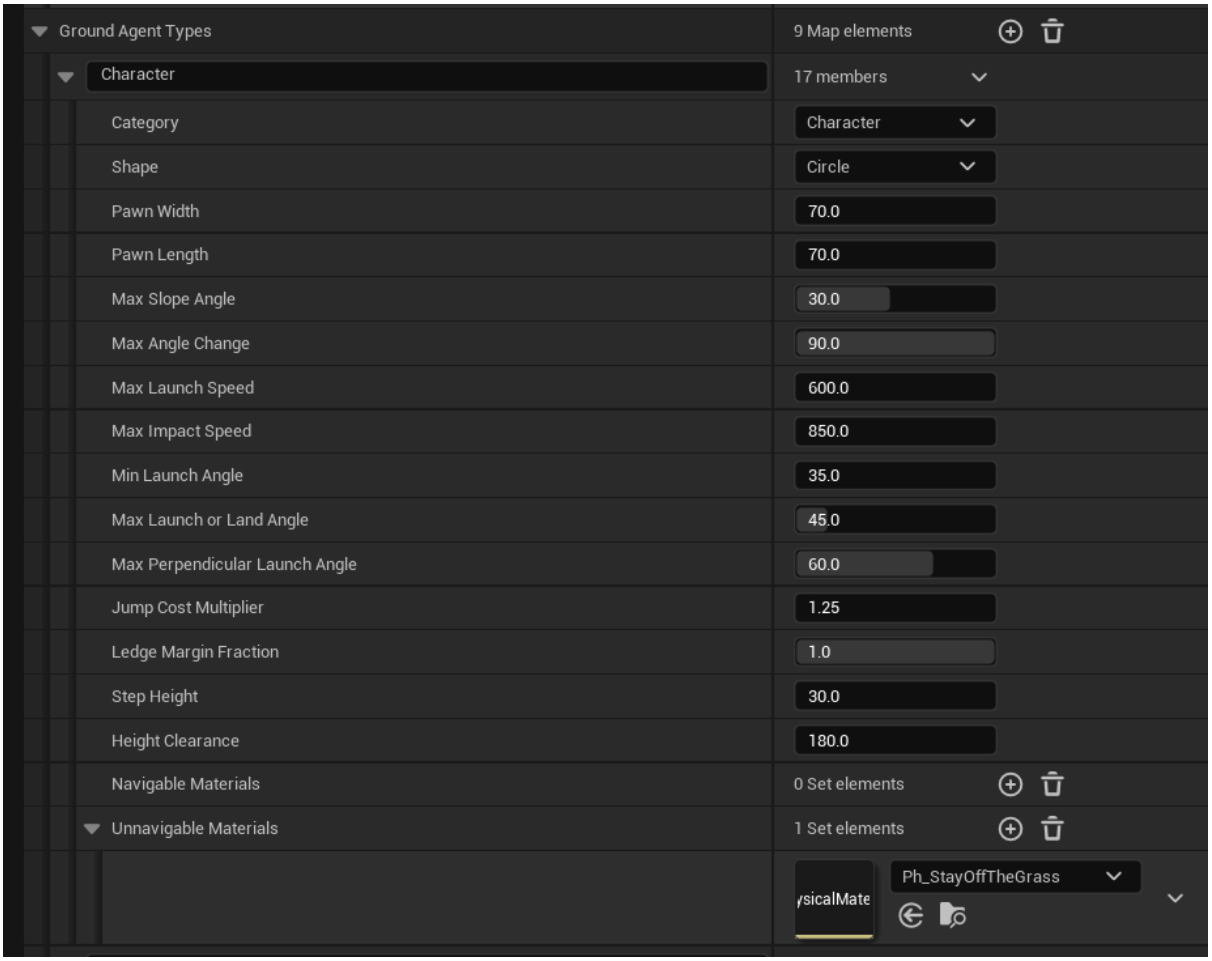
- **Binary evaluation** - The binary evaluation version of Mercuna must be installed as an Engine plugin - unzip the archive and copy the Mercuna directory to the **Plugins/Marketplace** directory (you may need to create the Marketplace subdirectory) within your Unreal Engine 5 install, e.g. *UnrealEngineDir*/Engine/Plugins/Marketplace/Mercuna.
- **Source version** - For source versions of Mercuna, it can be used either as a Game plugin or, if you are building the engine from source, as an Engine plugin. Simply copy the Mercuna directory into the **Plugins** directory in your Game/Engine folder.

If you have also licensed Mercuna 3D Navigation, then the plugin will provide both ground and 3D navigation capability. See the Mercuna 3D Navigation User Guide for documentation of the 3D Navigation capabilities.

Once installed, the Mercuna components, actors and menu will automatically be available when you next start the editor.

Configuring Agent Types for navigation

Mercuna Ground Navigation enables path finding and following for pre-configured agent types. There is no restriction on the number of agent types. The properties of these agent types must be set up on a per-project basis. In the UE4 editor, under **Edit | Project Settings… | Plugins | Mercuna** you will find **Ground Agent Types**. These specify the parameters for the different kinds of agents that will navigate using Mercuna Ground navigation.



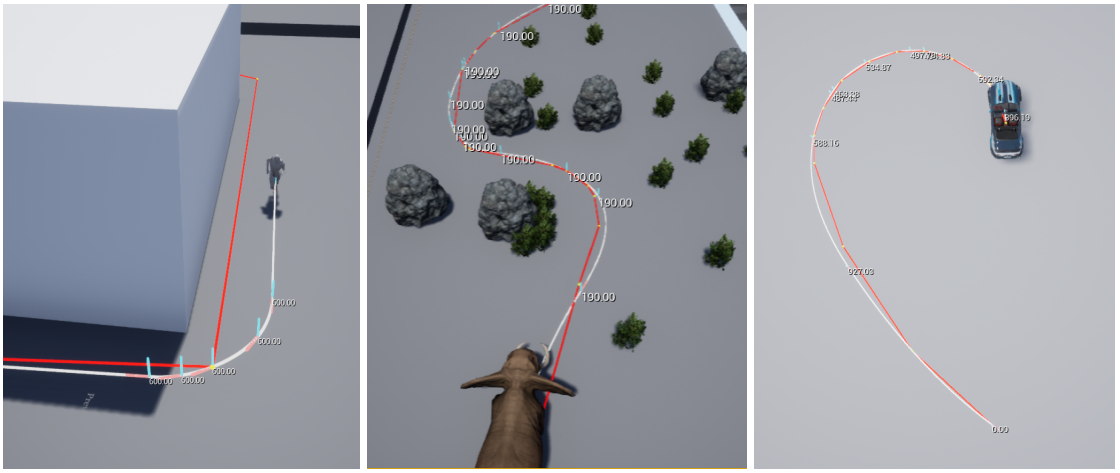
Configurable Agent Type properties

The properties of these agent types change the generation of the **Nav Grid** (see later) which marks the areas of the geometry over which your agent may travel. These parameters are:

- **Category [Character/Animal/Vehicle]** - The category of the agent. This determines the kind of paths the agent can follow and how it can move, more detail below.
- **Shape [Circle/Rectangle]** - The projected 2D shape of your agent, usually a circle for characters or a rectangle for animals and vehicles.
- **Pawn width** - How wide your pawn is (and hence how narrow a route you can path down). If the agent's shape is Circle, this gives the diameter of the agent.
- **Pawn length** - How long your pawn is (e.g. which destinations could your vehicle fit in lengthwise). If the agent's shape is Circle then this will automatically be set equal to the Pawn width.
- **Step height** - The height of the biggest step that an agent can step up or step down. This allows agents to go up stairs or make other steps between navigable surfaces that are at different heights.
- **Height clearance** - The height of your agent (i.e. the lowest tunnel your pawn can path through).
- **Max Slope Angle** - The maximum incline your pawn can ascend/descend (in degrees).

- **Max Angle Change** - The maximum slope angle between adjacent cells that should be considered navigable (reducing this prevents paths going over sharp peaks or troughs).
- **Max Launch Speed** - The maximum launch speed used when automatically generating jump links (controls how far and high an agent can jump).
- **Max Impact Speed** - The maximum impact speed on landing used when automatically generating jump links (controls how far an agent can fall).
- **Min Launch Angle** - The minimum upwards angle for jumps relative to the horizontal, used when automatically generating jump links. Increasing this causes automatically generated jumps to become steeper.
- **Max Launch or Land Angle -** The maximum upwards angle of either the launch or landing angle (whichever is smaller) relative to the horizontal. Jumps are rejected if both the launch and landing angle of the jump exceed this value. Decreasing this causes steeper jumps to be rejected.
- **Max Perpendicular Launch Angle** - The maximum sideways angle for jumps from an edge (relative to the perpendicular outwards from the edge), used when automatically generating jump links. Reducing this restricts the number of acceptable jumps.
- **Jump Cost Multiplier** - A configurable multiplier to make jumps less desirable for path finding. A value of **1** ensures that a jump has the same cost as a straight line path covering the same distance along a navigable surface. Higher values increase this cost proportionally.
- **Ledge Margin Fraction** - Distance to remain from ledges/precipices, expressed as a fraction of the half-width (for circular agents only). Reducing this allows characters to get closer to a ledge than they can to a wall.
- **Navigable Materials** - Physical materials that the agent is allowed to navigate on. If any materials are specified here then the agent can only navigate on those materials and no others. If no materials are specified here then the agent can navigate on any material except unnavigable materials.
- **Unnavigable Materials** - if no navigable materials are specified then the agent can navigate on any material except the physical materials specified here.

Agent Categories



The agent category controls whether the agent is allowed to turn on the spot, and how edges of physical materials with different friction are considered.

Characters are allowed to turn on the spot, so a standard A* pathfinder is used to find the shortest path to the character's destination regardless of how sharp the turns required are. Path smoothing is still applied so characters will avoid making sharp turns when there is space available to do so.

Animals prefer not to turn on the spot, so Mercuna's kinematic pathfinder is used. This means that animals will prefer to take corners in smooth arcs even if there is a shorter route that requires a sharp turn available. However, if there is no alternative the animal is able to turn on the spot.

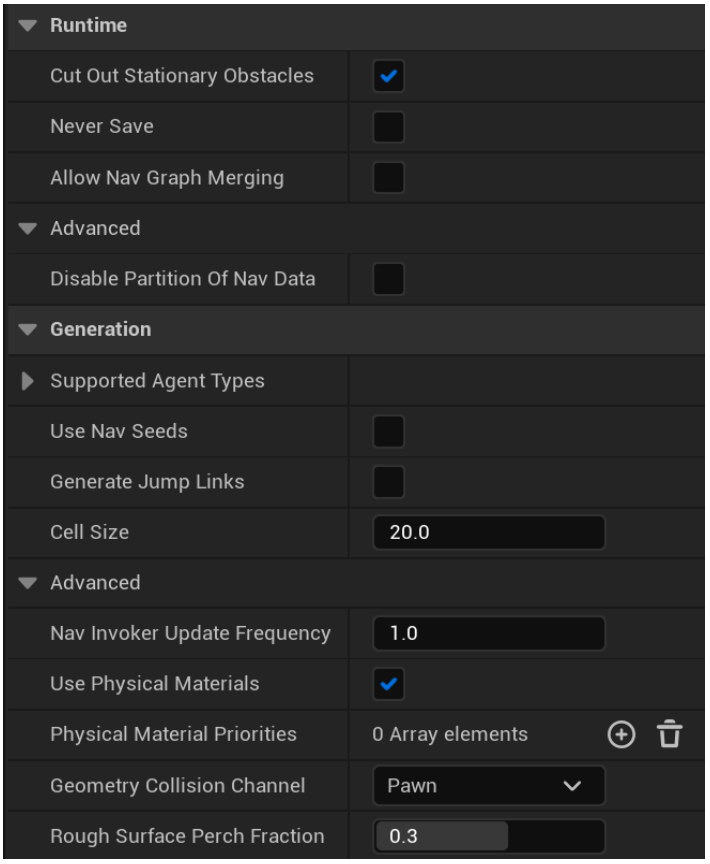
Vehicles can't turn on the spot. Mercuna's kinematic pathfinder is used, but if the only route to a destination requires a turn sharper than the vehicle's minimum turning radius, then no path will be found.

Additionally, for vehicles the available surface friction is considered at the sides of the agent shape, to account for the positioning of the vehicle's tyres.

The agent category also determines the way in which the navigation component for navigating pawns can be configured, presenting options appropriate for that type of agent.

Configuring the Navigation Grid

A **Mercuna Nav Grid** is used to find paths for characters, animals and vehicles.



Example settings for Mercuna Nav Grid

A Nav Grid is generated in the level everywhere that is within a **Mercuna Nav Grid Volume**.

Generation Settings

A Nav Grid can simultaneously support multiple agent types, controlled by the **Supported Agent Types** setting. Create agent types in the project settings, as detailed above.

Use Nav Seeds controls whether nav seeds are required. If switched off then unreachable areas aren't removed from the nav grid, increasing memory usage.

Generate Jump Links controls whether jump links should be automatically generated for this mesh. If true then an additional post-processing step is introduced, increasing peak memory usage during the build and increasing build times.

Cell Size sets the resolution of the Nav Grid which Mercuna will use to find paths for agents. Using a smaller cell size will allow a more accurate representation of the navigable space, but more memory will be consumed and pathfinds will take longer. As a guide, we suggest setting cell size to half the width of your narrowest agent.

If you have agents of very different sizes, it can be more efficient to use a second nav grid with a larger cell size for the larger agents. We recommend that the largest agent using a nav grid should be no bigger than three times the size of the smallest agent.

Note that agents must not be more than 63 cells wide or long. If an agent type is used with a nav grid that has too small a cell size, a warning will be logged and the cell size will be automatically increased.

Nav Invoker Frequency is only used if the nav grid is being used with nav invokers, and controls how often the nav grid should be updated as the invokers move around.

Use Physical Materials controls whether surface type information from Physical Materials is recorded in the Nav Grid. This information can then be used to guide or limit pathfind searches, as discussed below.

The **Physical Material Priorities** are discussed below.

The **Geometry Collision Channel** specifies the collision channel used to query whether geometry should be considered for navigation by this nav grid. For a collider to be considered by navigation it must have “Can Ever Affect Navigation” set to true, and give a Block response to this collision channel.

The **Rough Surface Perch Fraction** allows control of perching over rough surfaces, marking small non-vertical deviations in geometry navigable. The value is expressed as a fraction of the agent radius. Increasing the value causes the nav graph to cover a larger area, but may result in some areas being incorrectly considered navigable.

Runtime Settings

If **Cut Out Stationary Obstacles** is set then stationary obstacles are cut out of the nav graph so that paths will go around them. This stops obstacles from being treated as dynamic obstacles when stationary (see Stationary Obstacles below).

The **Never Save** property indicates whether the nav grid will be built from procedurally generated data at runtime, so any navigation data generated in the editor should not be saved (see Runtime Grid Building below).

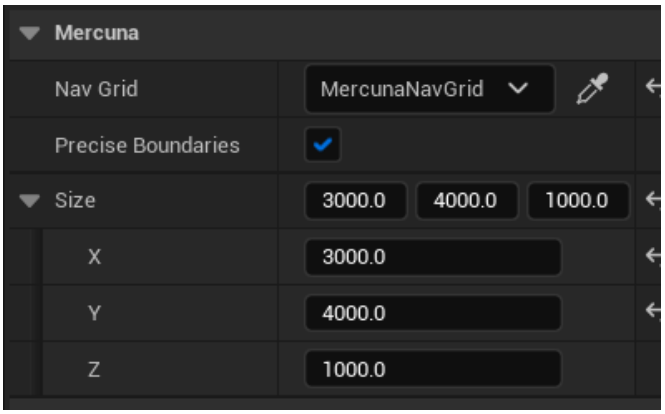
Allow Nav Graph Merging enables this nav grid to merge with other nav grids at runtime and allow continuous navigation between them.

Disable Partition of Nav Data (UE5 only) stops a nav graph being split up into chunks in world partition maps. This should only be used if you don’t want sections of the nav grid streamed in and out but want the entire nav grid always loaded.

Setting up the navigable space

To set up a nav volume, first add a **Mercuna Nav Grid Volume** actor into the level, and size it as required. Only boxes are supported, so to describe more complex boundaries of the navigable area, multiple Nav Grid Volumes can be configured.

If **Precise Boundaries** is switched on, the boundaries of the nav grid volume are considered hard edges, beyond which your agents can't move. However, if the Precise Boundaries option is turned off then navigable space will extend up to a high power of 2 boundary in the quad-tree. This allows a memory saving when there is open space around the navigable area and you don't need to precisely specify the edge of the area your agents will move within.



Example settings for Mercuna Nav Grid Volume

The grid generation parameters to be used in this level are configured on the Mercuna Nav Grid actor, and on the Agent Types set in the project settings. Upon creation, the parameters are set to default values. These defaults can be modified in the Mercuna project settings.

Nav Seeds

In order to identify which regions should be considered navigable, Mercuna requires you to place **Mercuna Nav Seed** actors into levels. This allows uninteresting regions, such as the small isolated areas inside hollow geometry or large areas outside of the level boundaries, to be excluded and avoids pawn positions getting clamped into disconnected regions.

A Mercuna Nav Seed needs to be placed on or above the ground in the area of the level where pawns will navigate. This seed is used during construction of the nav grid to find all connected reachable cells, by flood filling the region starting at the nav seed. Any area not connected to the seed will be considered unnavigable. If you have multiple disconnected areas in your level where you expect pawns to move, a seed must be placed in each separate area.

Nav Seeds are disabled if **Use Nav Seeds** is off on the Nav Grid, or when using Nav Invokers to generate the nav grid. In this case unreachable regions of the nav grid aren't culled, increasing memory usage - but this means that these regions can easily be connected by nav grid rebuild.

Note that if you also have Mercuna 3D Navigation, the nav seed will also seed any nav octree.

Exclusion Volumes

A particular box volume can be completely excluded from navigable space by adding a **Mercuna Nav Exclusion Volume** actor to the level. By default the exclusion volume will be applied to all nav grids (and all nav octrees if you also have Mercuna 3D Navigation).

If you have multiple nav grids in a level, and only want the exclusion volume to apply to one of them, then you can explicitly specify which grid the exclusion volume is linked to via the volume's properties.

The exclusion volume may be rotated and scaled.

Building the Nav Grid

Unless runtime generation is being used, the Nav Grid must be built after it is first configured or after the level geometry has changed. This can be done by selecting **Build Grid** from the Mercuna menu (accessed by clicking the Mercuna button in the Toolbar). An on screen notification displays the progress of the nav grid construction.

If full Mercuna logging is enabled (see Logging section below), then you will see the generation progress for each nav grid in the output log, and the total memory consumption of the grid is reported.

Commandlet

If you wish to automate the building of the Nav Grids, for example as part of a job run regularly each night, the grids can be built using the MercunaNavGraphBuilderCommandlet. This can be run from the command line using:

```
UnrealEditor.exe ProjectName.uproject MapName
-run=MercunaNavGraphBuilderCommandlet [-AlwaysBuild]
```

By default only grids that have been saved as dirty will be rebuilt and resaved, however you can force all grids to be built using the AlwaysBuild switch.

Memory usage and Performance

The main influences on memory usage and performance are:

- **Cell size**: Smaller cell size nav grids use significantly more memory and take longer to generate and find paths through.
- **Density of geometry**: Large open navigable volumes are stored efficiently, volumes with dense geometry and narrow corridors use more memory and take longer to find paths through.

Single Threaded Mode

When Unreal is running single threaded, Mercuna will only schedule jobs from the module Tick function. The budget per tick, specified in seconds, is set in the Mercuna section of Project Settings:

When in single threaded mode, nav graph queries may be time sliced over multiple frames. While rebuilding the nav grid works in single threaded mode, it is not recommended as you can easily schedule more work than it is possible to perform on a single thread.

Multiple levels

Mercuna supports both level streaming and world composition by saving the nav grid that is relevant to each streamed or composed level (sub-level) within that level.

When using level streaming or world composition, Mercuna Nav Grid Volumes and Mercuna Nav Seeds should be placed in the sub-level so that they are loaded and unloaded at the correct times.

When editing the Persistent level you will see one Mercuna Nav Grid in each loaded sub-level, this nav grid is automatically associated with the Nav Grid Volumes and Nav Seeds that are in that sub-level.

In order to allow seamless navigation between nav grids loaded from different sub-levels, Mercuna merges together nav grids loaded from sublevels at runtime. To enable this feature, the nav grids in the sublevels must have exactly the same settings, have the same orientation, and have the **Allow Nav Graph Merging** option set.

Additionally, all Nav Grid Volumes must have **Precise Boundaries** disabled in order to allow the associated nav grids to merge. This ensures alignment of the volumes present in each nav grid, so that whole nav grid tiles can be copied during nav grid merging.

If **Allow Nav Graph Merging** is false, the nav grid settings don't match, or precise boundaries are switched on, then nav grids won't be merged and multiple nav grids will exist at runtime. In this case you can manually switch pawns between nav grids using the **Set Nav Grid** function on the Mercuna Ground Navigation Component.

Generating Nav Grids in Multiple Levels

The best way to generate the sub-level nav grids is from the Persistent level. Load all your sub-levels using the levels window and then select Build All Nav Grids from the Mercuna menu. This ensures that geometry that overlaps between levels is correctly represented in each level's nav grid. Once generation is complete, save all the sub-levels.

If it is not possible to load all levels due to memory constraints, go through each sub-level containing a Mercuna Nav Grid in turn. Load a sub-level and all levels that contain geometry overlapping or close to Nav Grid Volumes within that sub-level, generate the nav grids in that sub-level by selecting them and using the Build Selected Nav Grid option from the Mercuna menu. Then save the sub-level and go on to the next.

World Partition (UE5 only) - experimental

Mercuna supports UE5's new World Partition system in a way that is transparent to the user and is designed to be simple to use.

In contrast to Level Streaming or World Composition maps, where a Nav Grid per sublevel is required, in a World Partition map you only need a single Nav Grid and then can freely place Nav Grid Volumes that can span large regions.

Internally Mercuna automatically splits the grid data into chunks and saves each part within an invisible grid of MerNavDataChunk actors. As these chunk actors are loaded and unloaded by the World Partition system, both in the editor and at runtime, the corresponding part of the grid will be merged in on load, or purged from memory on unload.

The whole map can therefore be covered by a single Nav Grid. Nav Grids are stored as non-spatially loaded actors and will always be present, whereas the Nav Volumes/Modifier Volumes/Exclusion Volumes are stored as spatial actors that are streamed in and out by the World Partition system as required.

Building

When the Nav Grid is built in the editor, only the part that corresponds to currently loaded editor cells will be generated. Actors immediately surrounding the loaded cells will be automatically loaded by the build process to ensure the edge of the generated volume is consistent. These actors are then unloaded once the build is complete. **You must not load or unload editor cells while the build is running**.

It is therefore possible to generate one part of the grid, save that section, and then load a different set of editor cells, generate the grid for that region, and then have both sections of the grid seamlessly merge together at runtime. However, we strongly recommend having an overnight job that builds the nav grid using the commandlet (see below) in order to ensure the nav grid is consistent with all changes that have been made to the map.

If the Nav Grid is placed into any data layers, then when building in the editor only actors that share at least one data layer with the Nav Grid will be included for the purposes of generating the grid.

Runtime builds and rebuilds work as normal, but note that they will only build based on the geometry that is currently loaded in the specified volume.

Nav Grids can also be built using the Mercuna world partition commandlet. This will build all the Mercuna nav graphs by iteratively loading the cells within the specified map. The commandlet can be run from the command line using:

```
UnrealEditor.exe ProjectName.uproject MapName
-run=WorldPartitionBuilderCommandlet
-Builder=MercunaWorldPartitionBuilder
```

Runtime

At runtime, as the world partition system loads and unloads actors as they come into range, the MerNavDataChunk actors will be correspondingly loaded/unloaded and merged/purged from their parent nav grid.

The merging and purging is done asynchronously so as not to cause spikes on the main thread. A consequence of this is that move commands to agents in the region may fail if executed too quickly as the nav grid data may not have yet been merged. This can most frequently occur if moves are executed on map BeginPlay. It can be resolved by adding a short delay before the move.

Load Complete Event

When a nav grid is loaded and ready for use the **OnLoadComplete** event on the Nav Grid triggers. In simple configurations where nav grids aren't loaded from multiple sub-levels simultaneously and there is no level streaming, this event is triggered immediately after the nav grid is loaded. However, if the nav grid is merged with another nav grid when the level is loaded, then OnLoadComplete is not triggered until the merge is complete. Once the event fires, you know that pawns are able to navigate across the merged part of the nav grid.

Runtime Grid Building

It can sometimes be desirable to build the Nav Grid at runtime, particularly for procedurally generated levels. Since these grids will be generated at runtime it is recommended that they should have the **Never Save** property set on them. This prevents any data that might be generated in the Editor while testing from being unnecessarily saved.

The build can be triggered by making a request on the nav grid actor (via Blueprint or C++) using the **Build** function. Once nav grid generation has fully completed, the event **OnBuildComplete** is triggered.

On Demand Grid Building

As an alternative to using Build to generate the whole of the nav grid, you can instead choose to generate only sections of the grid at runtime.

To do this, set **Never Save** on the nav grid as normal and then use the **Rebuild Volumes** function to build the grid in sections. If you want to generate the grid in volumes that may

have already been partially generated, but the geometry within those volumes has not changed, set **Only Unbuilt** on the call to **Rebuild Volumes**.

Spawning Nav Grid at Runtime

You can spawn nav grids at runtime in the usual way. The basic properties of the nav grid, such as cell size and agent type, must be set at spawn time. If you spawn through Blueprint then these properties can be set on the spawn actor function. If you spawn in code then you must use deferred spawning to set these properties before finishing the spawn.

You must then associate Nav Grid Volumes with the nav grid before building the nav grid using either Build, Rebuild Volumes or Nav Invokers. Use **Add To Nav Grid** on the Nav Volume to associate it with the nav grid at runtime. If this is done before the nav grid is built, then you don't need to call Rebuild Changes.

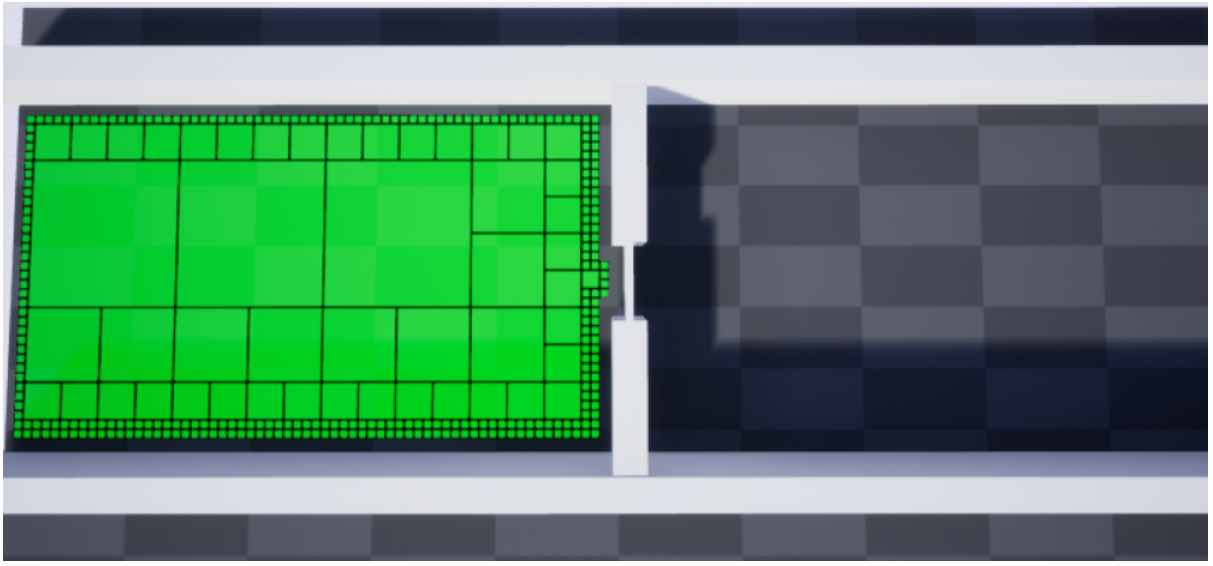
Runtime Grid Rebuild

Specific volumes of the Nav Grid can also be rebuilt while the game is running using the **Rebuild Volume** or **Rebuild Volumes** functions on the Nav Grid actor. Rebuild Volume is normally used to pick up runtime changes when just a section of the level changes, such as a door opening.

Once the volume has been rebuilt, the event **OnRebuildComplete** is triggered. This event includes the volume that was regenerated, to aid with scripting.

Pathfinds and reachability tests that go through the regenerated region may fail while the regeneration is in progress. Once regeneration is complete, all active paths are recomputed if they go through or close to the regenerated region, causing actors to path around new obstacles or through newly available shortcuts. If a pathfind fails during regeneration, you may want to retry it once the OnRebuildComplete event has been triggered.

Regenerating a region will not cause newly connected areas outside the regenerated region to become seeded. If a runtime regeneration might connect a volume that is not connected to any other nav seed when the nav grid is built in the Editor, you must place a seed within that volume.



In the screenshot above, the green region on the left is navigable and seeded. The wall in the centre has a door in it, and when this opens Rebuild Volume is triggered through blueprint to regenerate the navigation data around the door.

However, navigation into the region on the right will still not be possible because that region was not seeded. This can be fixed by adding a Mercuna Nav Seed into the right hand region.

Alternatively, if geometry might be destroyed in arbitrary locations so that you aren't sure what areas of the world might need seeding, you can disable nav seeds so that nav grid will cover all possible navigable surfaces regardless of reachability. To do this, switch off **Use Nav Seeds** on the nav grid.

Changing Nav Grid Volumes and Exclusion Volumes at runtime

Nav Grid Volumes, as well as Exclusion Volumes, can be spawned, moved or deleted after the nav grid has been built. The nav grid is not changed immediately, instead **Rebuild Changes** must be called to trigger a rebuild. Nav Grid Volumes must have the same orientation as the nav grid.

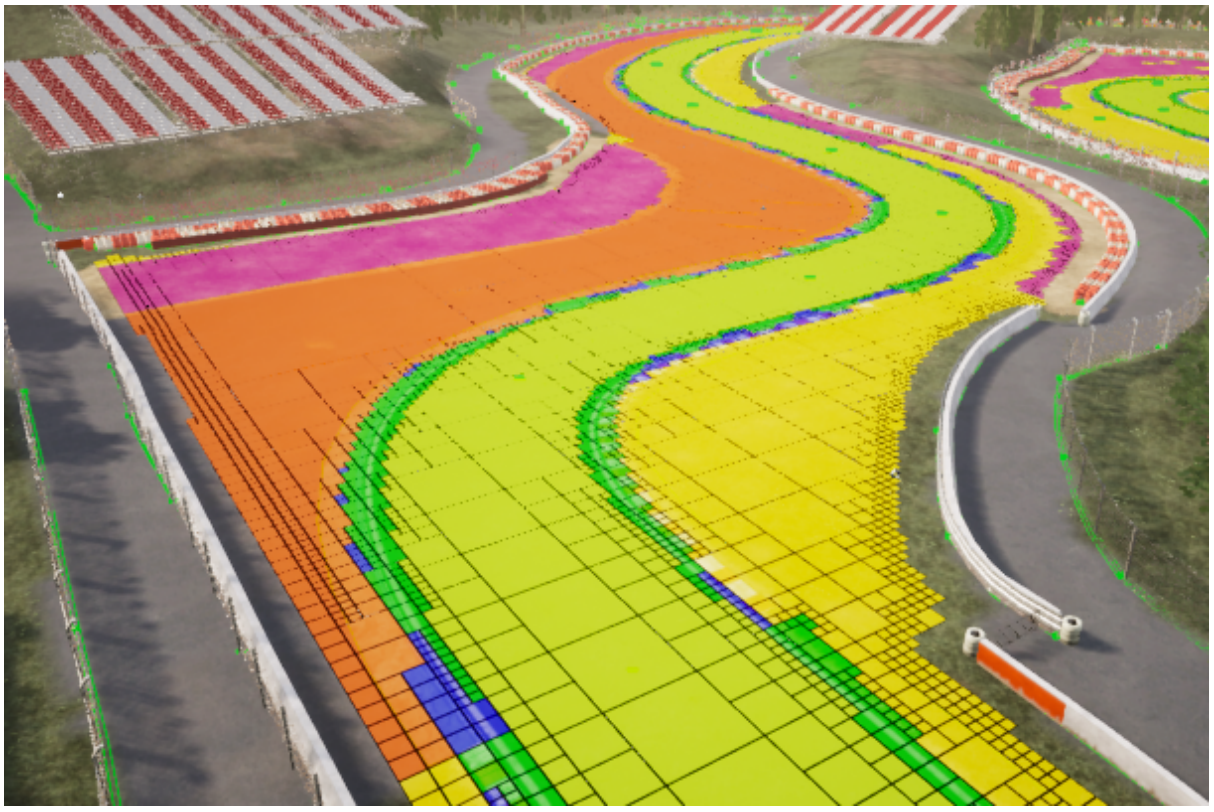
Changing Modifier Volumes at runtime

Nav Modifier Volumes can be spawned, moved, updated and deleted at runtime. The nav grid will automatically rebuild to reflect the changes.

Physical Materials

Physical Materials are used by Mercuna to determine the surface friction available for vehicles, as well as being one way of specifying regions of higher path cost (using a Mercuna Physical Material).

Therefore, it is important to check that the nav grid is picking up your physical materials correctly. You can do this with the **Cells - Surface Type** nav grid debug draw mode.



If meshes with different physical materials are overlapping, then the nav grid might pick the wrong one up during the build. To override which material is picked up, you can specify which physical materials have priority on the Mercuna Nav Grid actor.

Physical Material priority ordering

Any physical materials listed here will have priority in the order they are specified in the array, and all other physical materials will be given lower priority.

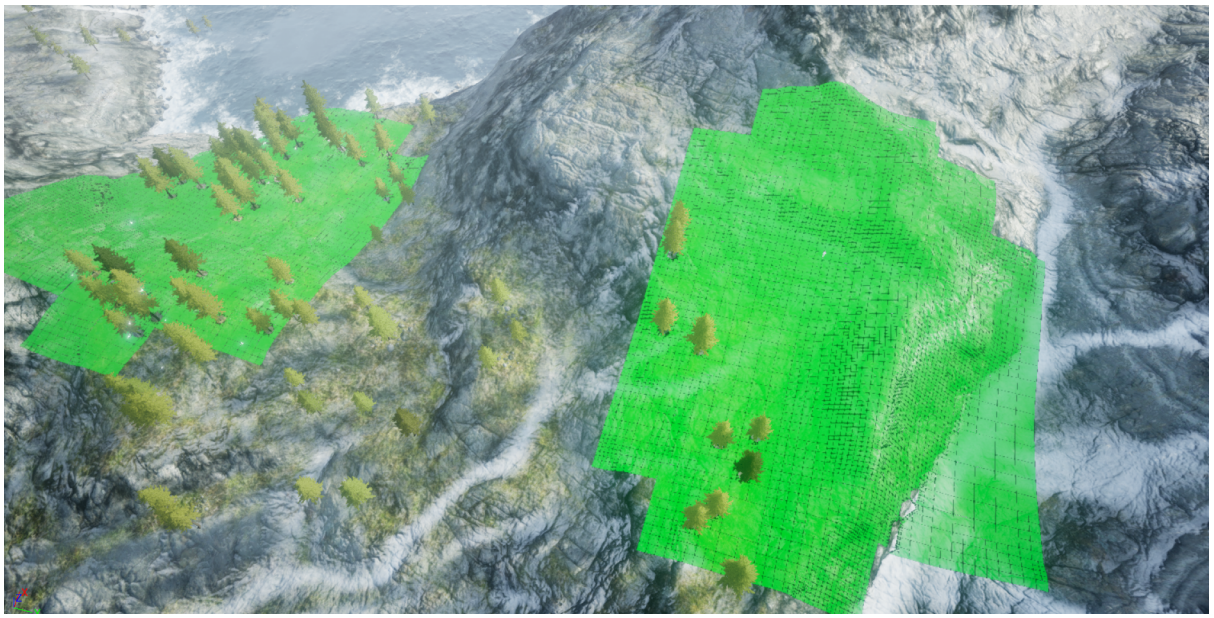
Physical Materials on Landscapes

Mercuna picks up physical materials from landscapes if they are set up, but the Unreal Engine configuration can be confusing. If you are seeing unexpected behaviour then these notes may help:

- Any physical material set on the **Landscape Material** on the Landscape is ignored. You must set the **Default Phys Material** on the Landscape instead.
- For landscapes with multiple layers, the physical material is set on the LayerInfo.
- In UE4 versions prior to UE4.25, there was a bug that meant physical materials weren't applied to the Landscape after being set. Workaround this by toggling the **Generate Overlap Events** setting in the **Collision** properties of the Landscape to cause UE4 to pick up the physical material change.

Nav Invokers

For procedural maps, or very large open worlds, it is sometimes not possible or desirable to generate the entire navgrid at editor or even at initial load time. Mercuna therefore offers a nav invoker system that generates and maintains patches of navgrid in a local area around specified actors. As these actors move through the world new navgrid is generated in front of them, and no longer needed navgrid removed from behind them.



Patches of navgrid created around two nav invokers

The advantages of using nav invokers is for very large worlds not having to store the entire navgrid in memory, and for procedural levels not having to wait at game start for the entire grid to generate. However the biggest disadvantage is the extra CPU usage as navgrid is constantly being generated and removed as invokers move around.

One other major constraint is that agents can only move to destinations within the area of the generated navgrid. Requesting an agent to move to a destination outside this area will fail.

Registering an invoker

Usually it will be AI/NPCs that are registered as nav invokers, though it is possible to register any actor as one. For each invoker you will need to specify the **Generation Radius** and the

Removal Radius. The Generation Radius is the distance within navgrid must always exist, while the Removal Radius is the distance with which navgrid can exist. Any navgrid beyond the latter distance from an invoker will be culled. However, for performance reasons these distances are approximated and not strictly obeyed. The Removal Radius must always be greater or equal to the Generation Radius. In cases where there exist multiple nav invokers of the same agent type within the world, navgrid is shared between them.

An actor can be registered as a nav invoker in two different ways.

1) Calling the **RegisterInvoker** function on the appropriate Navgrid. When registering a nav invoker through the RegisterInvoker function, the agent type must be specified manually. The actor can be unregistered using the corresponding **UnregisterInvoker** function

2) Adding a **Mercuna Nav Invoker** component to a pawn. If the pawn has a Mercuna ground movement component the invoker's agent type can be automatically detected from it. Alternatively, the agent type can be specified manually if required. When the pawn is destroyed it will be unregistered as an invoker.

In both cases, the Generation Radius and Removal Radius must be manually specified.

On registering the first invoker with a navgrid, if there are any existing areas of generated grid (that had been previously created via a runtime Build/Rebuild call) they will be cleared.

Runtime Rebuilds

When doing runtime rebuilding of an area of the navgrid, while using nav invokers, only parts of the grid within the specified volume that are within range of a nav invoker will be rebuilt. Also, only nav grid of the same agent type as the nav invoker will be rebuilt.

Invoker Volumes

Invoker volumes denote areas in the world that should be either always generated or generated in their entirety if any part of the volume is overlapped by a nav invoker. This can be particularly useful in geometry rich parts of the world, where a path to reach a nearby point may require the agent to take a detour outside of the currently navigable area.

An invoker volume can be created by adding a **Mercuna Nav Invoker Volume** actor to the level. Each invoker volume belongs to a nav grid and must have the same orientation as associated nav grid actors.

The invoker volume **Type** property can either be **Always** (default) or **OnOverlap**. A volume set to Always means that nav grid will always be present through the volume, regardless of whether any invokers are in or near it. Nav grid for OnOverlap volumes is only generated when the volume is overlapped by an invoker.

The nav grid for Always nav invoker volumes can be generated by an editor build. Just these volumes will be generated and then saved as part of the level. Building the grid for these

volumes at editor time saves having to generate the grid at game start up time. For building Always volumes in the editor, Use Nav Seeds must be disabled on the nav grid.

Pathfinding

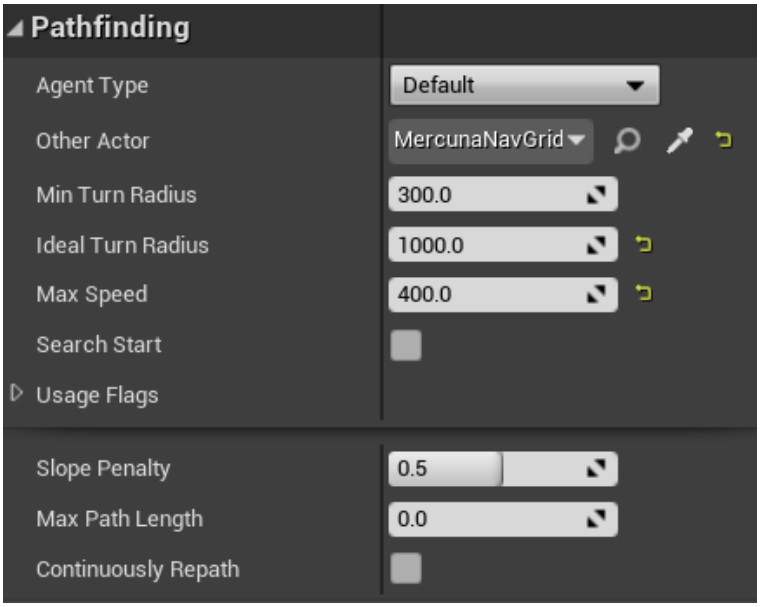
Finding paths through the Nav Grid can be done implicitly by making your pawn movement controlled by Mercuna, this method allows you to take advantage of the Mercuna steering system.

Mercuna also supports partial paths (disabled by default). A partial path is returned when a complete path can't be found to the specified destination, if it is disconnected from the start point, for example. Instead Mercuna returns a path to the closest point to the destination that is reachable.

Nav Grid Testing Actor

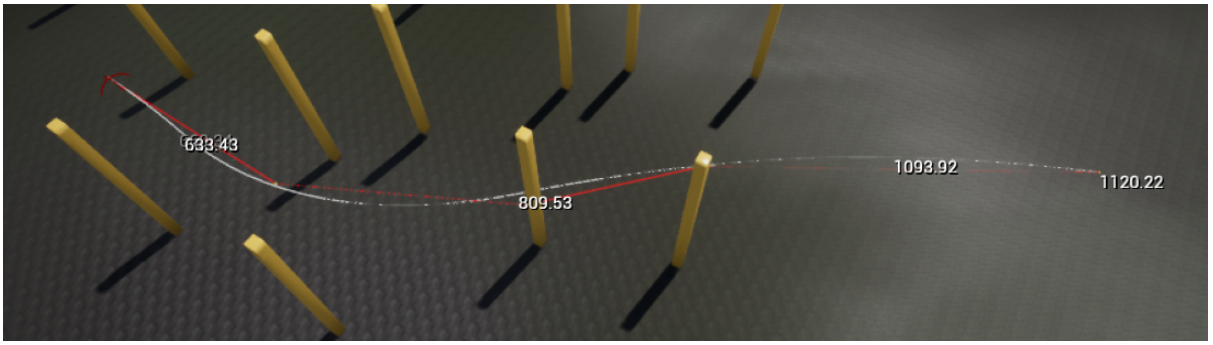
In order to easily debug and understand pathfinding problems a pair of **Mercuna Nav Grid Testing Actors** can be used to generate test paths. Simply drag two testing actors into the level, and on one of the actors set the other one as the **Other Actor**, and set the **agent type**.

When you do this a test path will be drawn connecting the two actors. This path will update when either actor is moved. A white path means a complete path could be found, while an orange path means that only a partial path could be generated.



The **Min Turn Radius** property sets the radius of the smallest turning circle, and the **Ideal Turning Radius** sets the turning circle it ideally tries to turn at if space is available. The **Max Speed** allows for some basic speed integration.

Slope Penalty (0-1) indicates how much the agent disfavours slopes (note that this is relative to the maximum slope for the agent type). **Max Path Length** allows you to control how far the pathfinder will search.

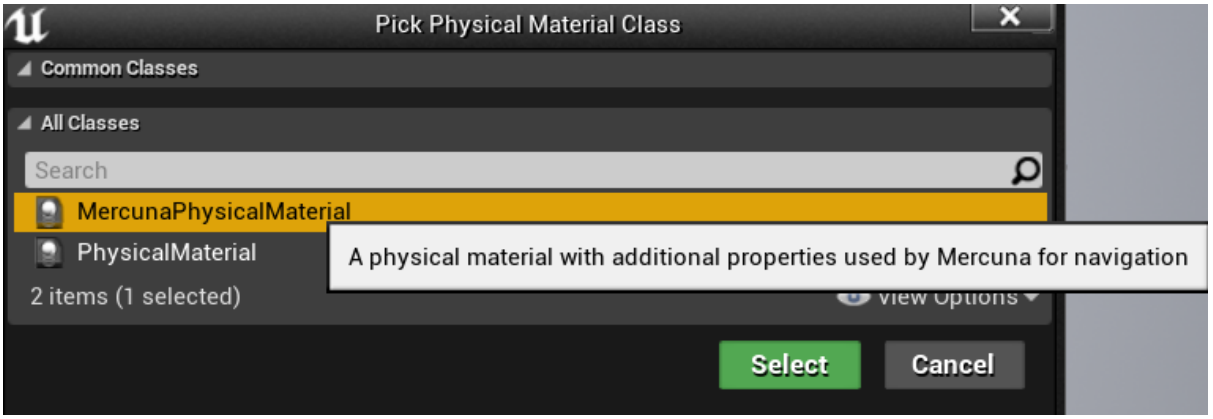


An example path between two Mercuna Nav Grid Testing Actors

Navigation Cost Multipliers

There may be particular surfaces that agents should not path over. Mercuna supports a custom Physical Material to allow this to be specified.

To create a Physical Material with a navigation cost multiplier, create a new Physical Material blueprint and specify **MercunaPhysicalMaterial** as the class.



Creating a Mercuna Physical Material

The Mercuna Physical Material has an additional property, Cost Multiplier.

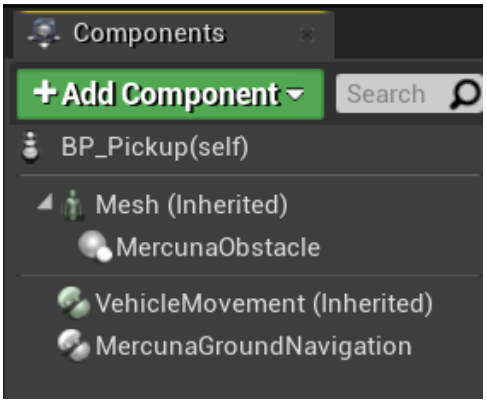
Mercuna will treat paths over any surfaces using this material as if they were **Cost Multiplier** times longer than their path length. This means if the multiplier is set above 1.0x then Mercuna will prefer paths that navigate around the surface with this material.

Due to the nature of the A* algorithm used for pathfinding, costs can only be increased and not decreased below 1.0x. Using a very high cost multiplier will mean that the pathfinder will search a long way for alternatives, and thus can considerably increase the computational cost of the path find. For this reason, the maximum cost multiplier that can be set is limited to 15.0x.

Creating a Mercuna navigated Agent

In order to allow pawns to use Mercuna to navigate they need to have the following components:

- **Mercuna Ground Navigation** - this component provides the pawn with navigation capabilities, accessible through Blueprint.
- **Mercuna Obstacle** - this marks the pawn as a dynamic obstacle for the purpose of ground navigation. The obstacle component must be a child of the root scene component.
- A suitable **movement component**, currently this must be either a **Character Movement** or **Wheeled Vehicle Movement** component



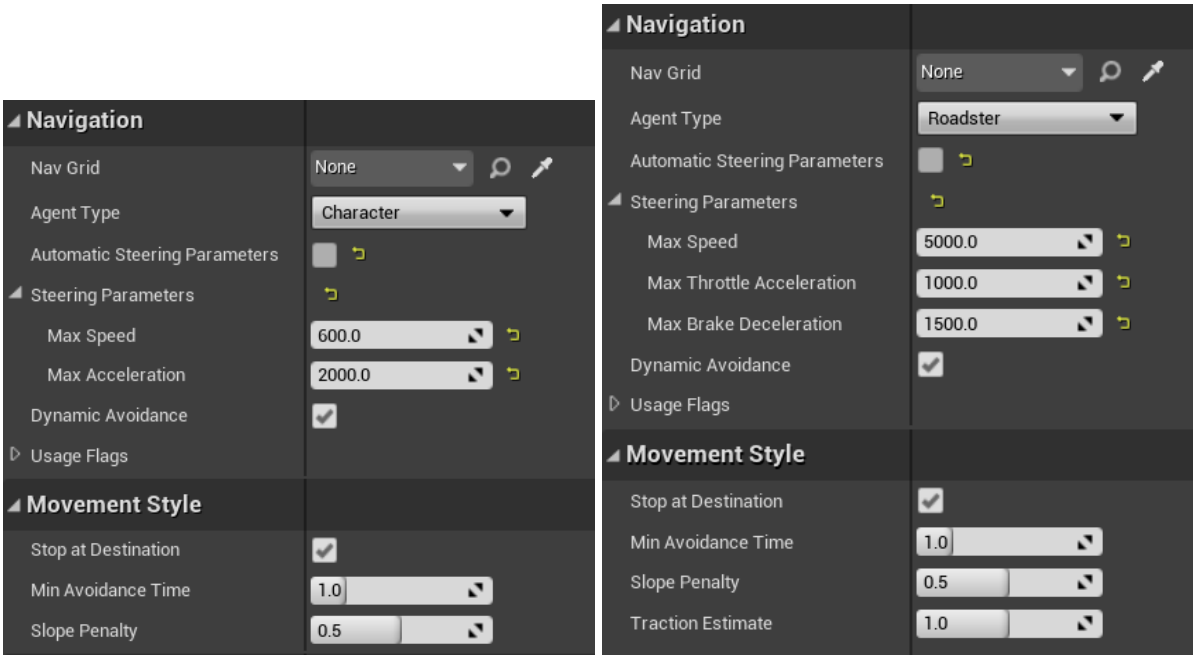
Vehicle blueprint setup for Mercuna navigation

Navigation Component

The navigation component offers both a C++ and Blueprint interface for making movement requests to the pawn.

Mercuna will try to automatically find a Mercuna Nav Grid that is compatible with the **Agent Type** set on the Ground navigation component, and automatically fill in **steering parameters** based on the setup of the pawn. Alternatively, these can be explicitly overridden here.

For the purposes of navigation the size and shape of the agent will be taken from the Agent Type. When the navigation component is selected the agent's navigation shape will be drawn in the viewport so that you can check that it is a good fit for the agent.



Example ground navigation components where steering parameters have been explicitly set

Example for a wheeled vehicle

For wheeled vehicles, check these settings in your agent type configuration:

- The **maximum slope angle** (in the project settings) - the vehicle may 'believe' that it can travel up slopes that it cannot, or cannot travel up slopes it is able to.
- The **step height** or **max angle change** is too high - The vehicle attempts to drive over terrain that is too rough

If you are manually setting the steering parameters that control how Mercuna expects the vehicle to behave, then

- **Max Speed -** The maximum speed of your vehicle
- **Max Throttle Acceleration** - The acceleration of your vehicle, limited by the power of your engine. Note that for high values you may be limited by the friction of the physical surface.
- **Max Brake Deceleration** - The deceleration provided by your brakes.

Specific to Vehicles, in the Movement Style there is the **Traction Estimate**, increasing this above 1 causes the vehicle to overestimate the available traction and make higher speed turns at the risk of sliding. Reducing the value has the opposite effect of underestimating the traction resulting in more cautious driving.

The reverse speed will be picked up automatically for PhysX vehicles, and for a NavGridTestingActor the maximum speed in reverse will default to ¼ of the MaxSpeed.

Example for a slowly turning animal (e.g. a horse)

Mercuna ground navigation can be used to steer an agent that has a CharacterMovementComponent.

It is recommended to use velocity-based (rather than acceleration-based) path following (this is due to the UE character movement component rotating the character based on the acceleration vector input). To configure velocity-based path following make sure that **Use Acceleration for Paths** is disabled on the CharacterMovement component.

Either set **Orient Rotation to Movement** True, or to get smoother rotation set:
Use Controller Rotation Yaw False on the Character, and set **Use Controller Desired Rotation** True and **Orient Rotation To Movement** False on the CharacterMovement Component.

If you are manually setting the steering parameters that control how Mercuna expects the animal to behave, then

- **Max Speed -** The maximum speed of your animal
- **Max Acceleration** - Should be set to the maximum acceleration of your animal
- **Min Turning Radius** - Radius of the tightest turn your animal can make while moving

Example for a humanoid Character

Settings should be similar to a slowly turning animal. By setting the agent category to Character the pathfinder will allow paths with sharp corners, taking advantage of the characters ability to change direction quickly.

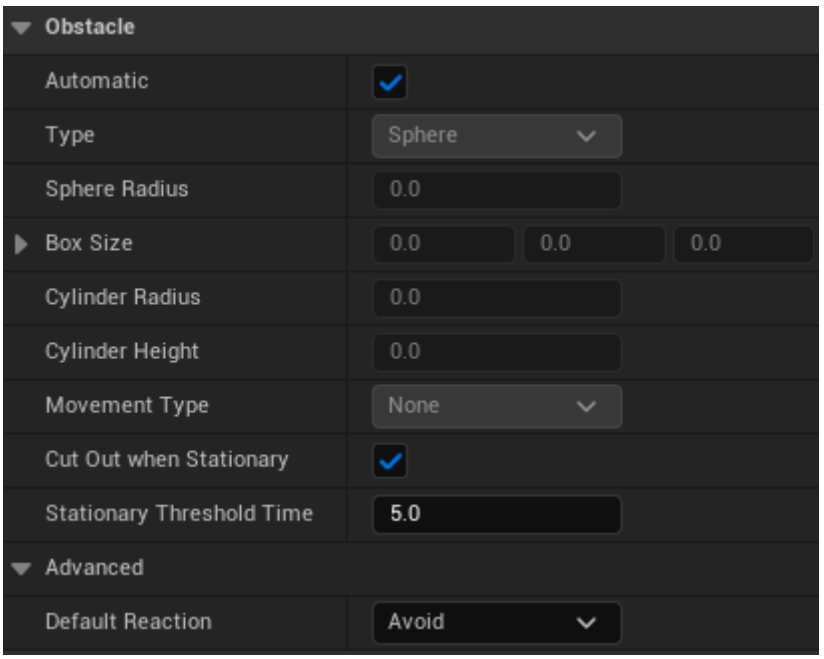
Updating parameters during the movement

A subset of the steering parameters can be updated even whilst the agent is moving, for example to make a character follow a path more or less aggressively, via the **UpdateDynamicSteeringParams** function from C++ or Blueprint. These parameters can be updated frequently but should not be changed every frame. These changes will persist even after the move has completed.

To change only the maximum speed for the duration of the current move, the **OverrideSpeedMultiplier** function can be used.

Obstacle Component

Any actor can be made a dynamic obstacle by adding a **Mercuna Obstacle** component. It is expected that Mercuna navigated pawns will usually have one, but other pawns such as the player, might also have an obstacle component so that the AIs avoid them.



Obstacle settings

By default Mercuna will automatically try to calculate the best shape of obstacle that encloses the pawn, for pawns with a **Mercuna Ground Navigation** component this will be set based on the agent type. Alternatively, the shape and size can be manually set. The available shapes are Sphere, Box, Cylinder. The shape only applies for avoidance.

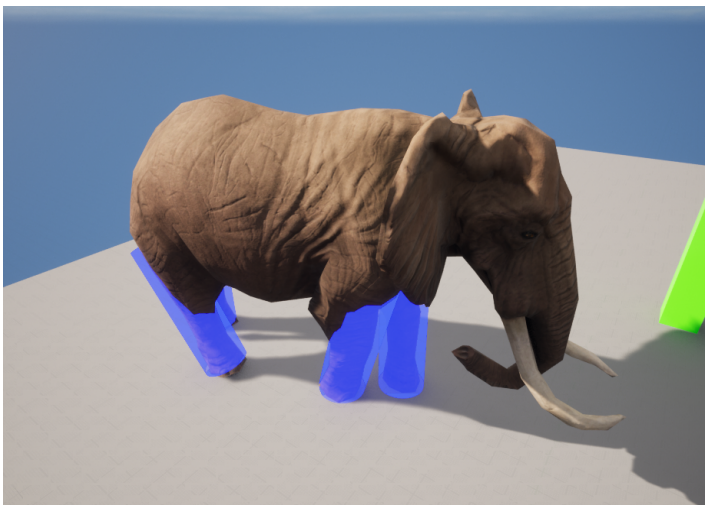
The **Movement Type** configuration specifies how avoidance should treat the agent. Characters will prefer to deviate from their path in order to avoid other agents, whereas Vehicles will prefer to adjust their speed but stay on their path. Additionally, characters will adjust their paths assuming that vehicles will not avoid them.

Actors with an Obstacle component should have “Can Ever Affect Navigation” switched off on their Collision settings on all their components so they aren’t baked into the nav grid.

The **Cut Out When Stationary** and **Stationary Threshold Time** options configure whether the obstacle is cut out of the Nav Grid when it is stopped, if this option is enabled on the Nav Grid. See Stationary Obstacles for more detail.

The **Default Reaction** parameter specifies whether other navigating agents will normally avoid this agent. For more fine grained control, avoidance of a particular obstacle by a particular agent can be configured using the SetAvoidanceAgainst call from C++ or Blueprint.

It can sometimes be desirable to use multiple obstacle shapes to describe an actor, rather than just using a single large obstacle shape. An example of this might be a large animal that wants to allow smaller animals to run between its legs. In this situation you can add multiple obstacle components and attach one to each leg using a socket.



An elephant which each leg setup as a separate obstacle

Modifier Volumes

Nav Modifier Volumes provide a mechanism for designers to influence and limit navigation within specific regions. For example, you can increase the cost of navigating through an area, such that paths will prefer to go around it, or you can mark volumes as being restricted to particular pawn types, or pawns as restricted to staying within certain volumes.

Usage flags

In the project settings you can define up to 32 custom global usage types. These can then be assigned to individual Modifier Volumes to mark what the volume represents. Flags corresponding to each usage type can be set on the Ground Navigation Component of pawns to indicate which types of volumes it is allowed to enter, or that it is required to stay within.

For example, if you define a Fire usage type, you can mark Modifier Volumes as representing a region that is on Fire. By default pawns will not be allowed to enter that region, but they can be configured to enter and move through Fire regions as normal, or alternatively can be restricted to only move inside of Fire regions (e.g. a fire demon).

Costs

During a path search through the Nav Grid, Modifier Volumes allow designers to increase the cost of specific volumes and therefore discourage pawns from moving through them.

When a volume has a modified cost multiplier, the path distance is multiplied by the cost multiplier to calculate the expense of traversing a volume. The size of the additional cost of passing through a volume determines how far pathfinding will search for longer alternative routes that avoids it, before deciding to use it as part of the path.

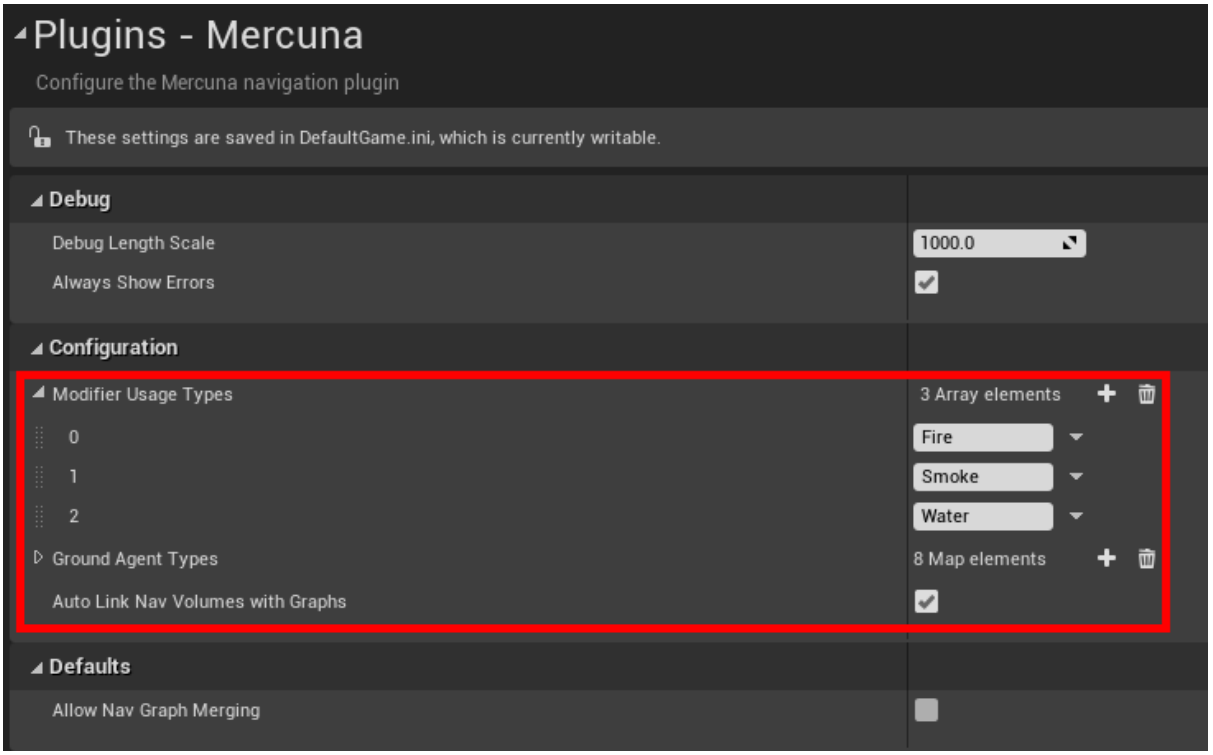
Due to the nature of the A* algorithm used for pathfinding, costs can only be increased and not decreased below 1.0x. Using a very high cost multiplier on a volume will mean that the pathfinder will search a long way for alternatives, and thus can considerably increase the computational cost of the path find. For this reason, the maximum cost multiplier that can be set on a volume is limited to 15.0x.

If multiple modifier volumes overlap, the maximum cost multiplier across the overlapping volumes is used.

Using Modifier Volumes

Defining usage types

Usage types are defined in the Mercuna page of the project settings. Up to 32 usage types may be defined.



Warning: removing usage types doesn't update the usage flags configured on existing modifier volumes or navigation components. Removing the Water usage type in the above example, means that all the usage flags on actors that previously corresponded to Water, will now be applied to the Glue usage type instead.

Creating modifier volumes

To create a modifier volume, add a **Mercuna Nav Modifier Volume** actor into the level, size and rotate it appropriately. Only box shaped Modifier Volumes are supported. To describe more complex boundaries, multiple Nav Modifier Volumes can be created.

Configuring modifier volumes

In the Mercuna section of the modifier volumes property panel, the cost and usage types for this volume can be configured.

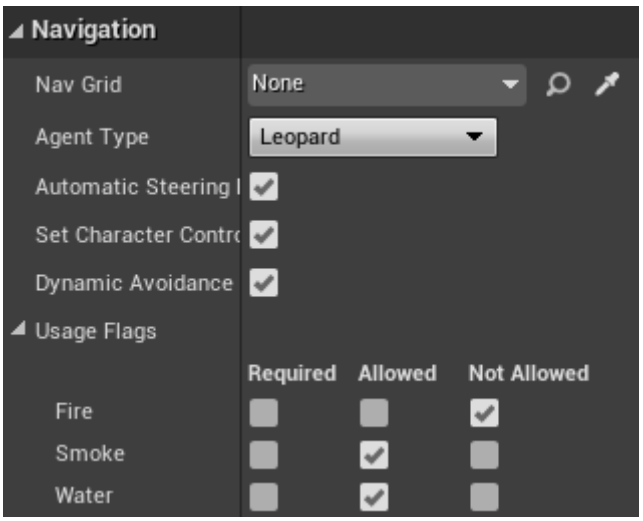
Modifier volume properties

Whenever a volume is created, moved or resized in the Editor the navigation grid must be rebuilt for the change to be applied.

Nav Modifier Volumes need to be associated with a nav graph. Similarly to Nav Grid Volumes, if the Mercuna project setting “Auto Link Nav Volumes with Graphs” is enabled (the default) new modifier volumes will be automatically linked to the nav graph if there is only one nav graph in the level when they are created. Otherwise you need to manually set the correct Nav Grid in their properties.

Configuring navigation components

Costs are automatically taken into account during pathfinding and spatial searches, but by default agents will not enter modifier volumes that have any usage types set. To allow or require pathfinding through volumes of particular types, the corresponding usage flags can be set on the Mercuna Ground Navigation Component.



The default for each flag is Not Allowed. Required means the agent can **only** move in volumes and through nav links with that usage type set. Allowed means the agent may, but does not have to, enter volumes and nav links with that usage type.

Runtime changes

At runtime, the settings on Modifier Volumes can be updated using the **SetEnabled**, **SetUsageFlags** and **SetCostMultiplier** functions on the Modifier Volume actor. These functions are exposed to blueprint.

Any paths through the changed modifier volume will be updated to reflect the change.

Changing these settings is relatively inexpensive, compared with the cost of a full grid rebuild for the modified volume.

Modifier volumes may be added, removed or moved at runtime, however this requires rebuilding the nav grid in the modified volume (the rebuild is triggered automatically), so should not be done too frequently.

To add a modifier volume, spawn a Modifier Volume actor, set the usage types and cost multiplier, and then use the **AddToGraph** function to add the modifier volume to the nav grid.

To remove a modifier volume, simply destroy the actor, or alternatively call **RemoveModifierVolume** on the NavGrid.

Modifier Volumes can be moved, rotated and resized by changing their transform in the normal way.

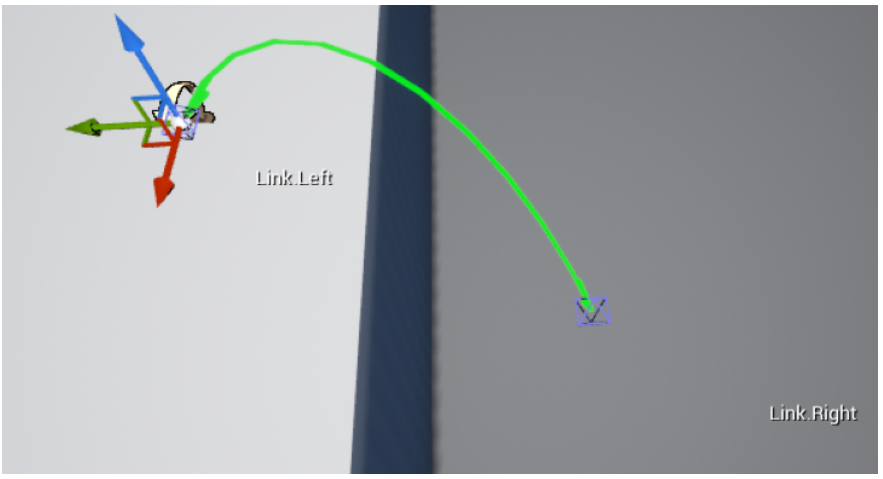
Restrictions

A maximum of 32 modifier volumes can be present in each 64x64 cell tile of the nav grid. There is also a limit of 255 distinct overlapping combinations of modifier volumes within a tile.

Nav Links

Sometimes you will want agents to navigate in non-standard ways, for example climbing a ladder or making a jump. Mercuna Ground Navigation supports this through Nav Links.

A Nav Link has two ends, labelled Left and Right, and a connection is inserted into the navigation graph between the two ends when the nav grid is built. You can choose for the link to be navigable in one or both directions.



A simple navlink representing a jump

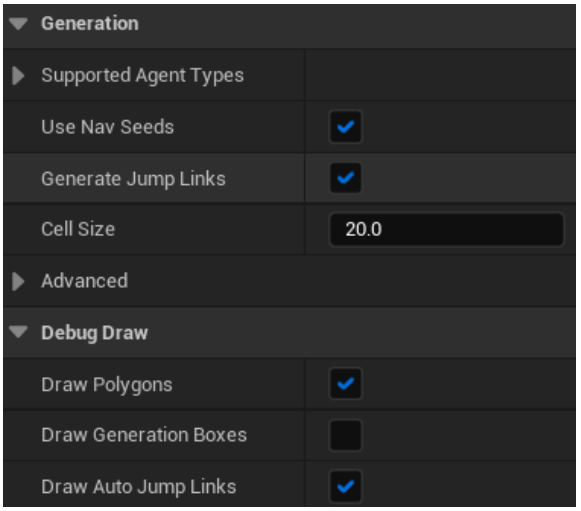
Mercuna supports both automatic generated and manually placed Nav Links. Automatic links cover basic jumps between navigable geometry, based on kinematic constraints configured for a particular agent type.

Manual links are placed by you and can cover more non-standard navigation more generally. Your Blueprint or Game code is responsible for actually traversing a manual link, whether by teleportation, playing an animation or any other method desired. Once the agent reaches the end of the link, Mercuna takes control again and continues the navigation.

Automatic Nav Links

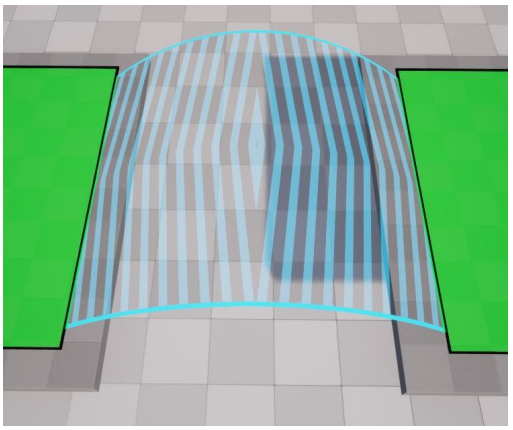
Nav Link Generation

Mercuna can automatically find and place nav links for jumps when building the nav grid. Automatic jump link generation can be enabled by toggling **'Generate Jump Links'** in the **Generation** options for a **MercunaNavGrid** instance.

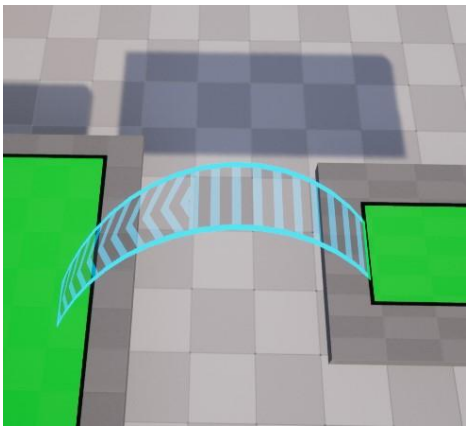


Once enabled, jump links are generated automatically the next time the grid is built.

Automatic Jump Links are created as rails between two polymesh edges, or a polymesh edge and a navigable surface. These links can be either bi-directional or one-way, depending on the height difference between the start and end, as indicated by the presence of arrows on either side of the link, with a green start-bar for the side(s) that can be entered.

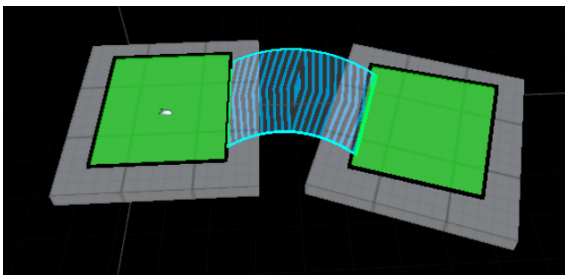


A bi-directional Automatic Jump Link

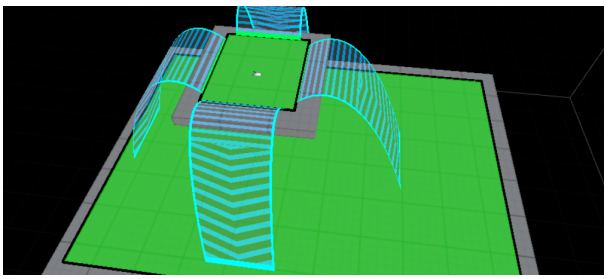


A one-way Automatic Jump Link

Automatic Jump Links can either be from an edge to another edge, in which case their maximum angle is defined by the user configured **Max Perpendicular Angle**, or from an edge to a lower surface, in which case they are projected perpendicular to the original edge.



Automatic jump from edge to edge



Automatic jumps from edge to surface

Display of the links can be controlled via the Draw Auto Jump Links option on the **MercunaNavGrid** actor, and the colour can be edited in Editor Preferences > Plugins > Mercuna > Auto Jump Link Color.

Automatic Jump Links are only created during editor builds and runtime builds. They are not created in runtime rebuilds or navgrid generated by nav invokers. In the latter cases the new navgrid will be generated without jump links and any jump links that previously existed in that region will be removed.

Configuring Automatic Jump Links for an Agent Type

Parameters controlling the Automatic Jump Links are configured on a per agent type basis and can be found in the Agent Type Settings. Parameters relevant to Automatic Jump Links are:

- **Max Launch Speed**, which determines the maximum height of an agent's jump.
- **Max Impact Speed**, which determines the maximum distance an agent can fall.
- **Min Launch Angle**, which determines the minimum upwards angle of an agent's jump relative to the horizontal.
- **Max Launch or Land Angle**, which determines the maximum upwards angle of either the launch or landing angle (whichever is smaller) of the agent's jump relative to the horizontal. Jumps are rejected if both the launch and landing angle of the jump exceed this value. This ensures, for example, that steep jumps between edges at similar height are rejected but drops from ledges to the ground (which have a high landing angle but a low launch angle) are kept.
- **Max Perpendicular Launch Angle**, which determines the maximum sideways angle of an agent's jump relative to an edge.
- **Jump Cost Multiplier**, which determines how expensive jumps are considered by path finding, relative to a straight line path along the ground covering the same distance.

The agent's size is considered by Automatic Nav Link generation and links are only generated where there is sufficient space for the agent to clear any obstacles. Links are also disallowed if they fall below minimum length, or connect edges which are trivial to navigate without jumping.

Manual Nav Links

Nav Link Creation

Normally, you will create a blueprint for each type of nav link supported in your game, and then place instances of these blueprints into your levels.

To create a Nav Link blueprint, add a new blueprint class, deriving from MercunaNavLink.

When an agent uses the nav link, the **Nav Link Start** event is triggered on the nav link. This supplies the Pawn navigating the link the direction the link is being navigated (left to right or right to left) and the position of the opposite end of the link.

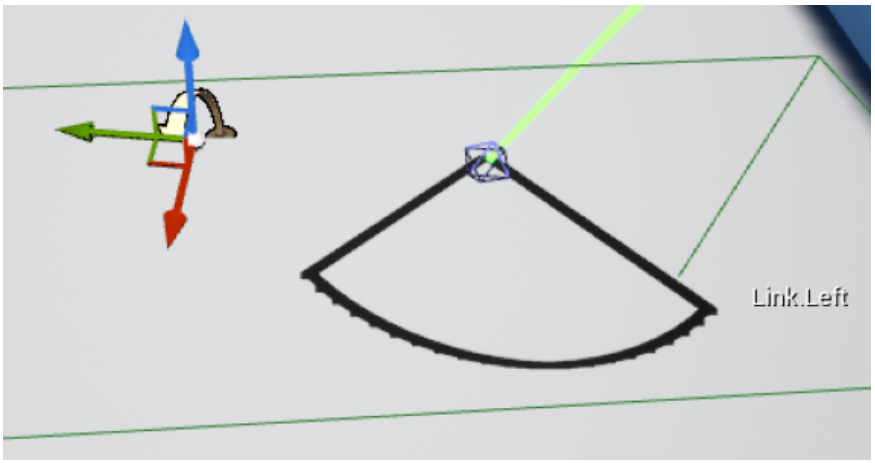
When this event is fired your blueprint or game code should cause the agent to move to the destination, e.g. by triggering an animation.

To complete the nav link and resume the agent moving down the rest of the path, you can call the **Nav Link Complete** function on the **Mercuna Ground Navigation** component. Alternatively, you can set a non-zero auto completion distance for the destination end of the nav link, if this is set Mercuna will automatically trigger link completion once the agent is within that distance of the destination.

Configuring the Nav Link

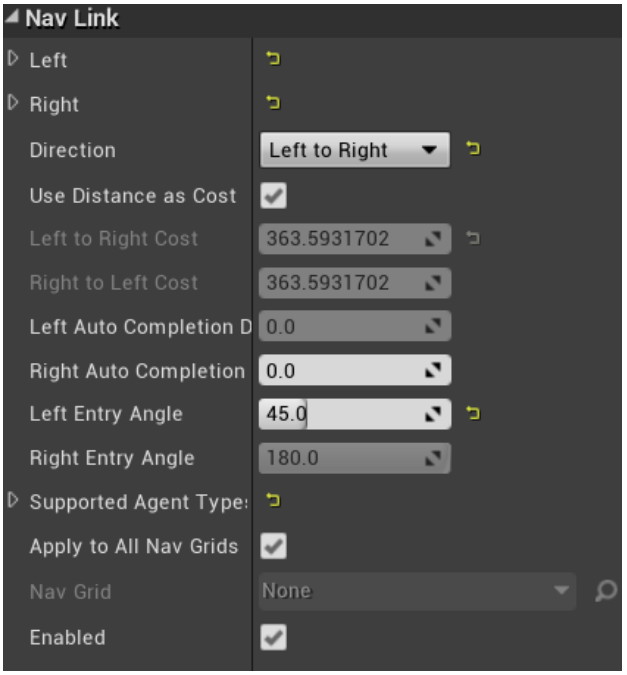
After selecting the nav link actor, you can position the ends of the nav link by clicking on them and then moving them in the normal way.

If you specify a limited **Entry Angle** range then you can also rotate the end to determine the acceptable range of entry angles, the black arc represents the valid entry angles.



Cost

The path finder normally considers the cost of moving through the navigable space to be equal to the distance the agent needs to move. For nav links, the cost can be adjusted to make agents prefer to use the link, or prefer not to. By default, the cost is set to the distance between the endpoints.



Note that if the link has a lower cost than the straight line distance, then it will only be used preferentially if the path finder discovers the link during its search. This won't happen if a straightforward path is

available, but the agent would have to go backwards to find the cheaper nav link, for example.

Usage Types

Similarly to Nav Modifier Volumes, usage types can be configured on Nav Links in order to restrict which types of agents can navigate the link. See the Modifier Volumes section of the documentation for full information on setting up Usage Types.

Note that if an agent has a usage type set to Required then that usage type **must** be set on the nav link for it to be able to use the nav link.

Agent Types and Nav Grids

You must specify which agent types the nav link supports. By default, the nav link will be associated with all nav grids that have navigable space at the endpoints of the link, if you want to restrict it to a particular nav grid, you can specify that explicitly.

Steering

In order to give smooth, natural looking movement Mercuna uses polynomial curves to interpolate between path points to generate smooth curves. These splines are constructed taking into account the available space.

Whereas the pathfind is performed when an agent is given a destination, the smooth curve is generated on demand. If the agent is pushed, or has to avoid other actors causing it to move away from the curve, then the curve and, if necessary, the underlying path are automatically regenerated.

Avoidance

Mercuna offers dynamic obstacle avoidance to ensure that pawns don't collide while moving. Any actor with a **Mercuna Obstacle Component** is automatically considered as an obstacle that needs to be steered around for the purpose of avoidance. For the purpose of avoidance agents will be treated as circles if the obstacle is a sphere or a cylinder, or as a rectangle for box shaped obstacles such as vehicles (if the rectangle is rotating rapidly it is treated as a circle).

Mercuna offers two different dynamic avoidance methods:

- **ORCA** - ORCA is an efficient linear velocity obstacle method that seeks the nearest non-colliding velocity. It is better for agents with independent destinations.
- **Context Steering** - Contextual Steering is an avoidance method that balances collision avoidance with soft constraints such as maintaining medium-range separation. It is good for teams of agents likely to move consistently.

The **Avoidance Mode** an agent uses can be configured on the Mercuna Ground Navigation Component.

In addition, the **Avoidance when Stationary** property chooses what to do when no command (i.e. no path) is present. The options are:

- **None** - i.e. don't move out of the way of nearby agents
- **CollisionOnly** - Only move to prevent an obstacle collision
- **Full** - Prevent collisions and for Context Steering apply cohesion & repulsion with nearby agents (for ORCA this is the same as CollisionOnly)

ORCA

Mercuna's implementation of Optimal Reciprocal Collision Avoidance (see e.g. https://gamma.cs.unc.edu/ORCA) is a modified velocity obstacle method that additionally takes into account the fixed level geometry stored in the nav grid.

Min Avoidance Time - Can be adjusted to change how early the agents attempt to avoid a collision (shorter times mean the agents move past each other more aggressively). A default avoidance time of around 0.3s is recommended for characters, or 1s for vehicles.

Context Steering

Contextual steering allows agents to perform path following whilst prefering to maintain pre-specified distances from other agents. Through combining the output of various steering contexts each agent attempts to choose a velocity that best satisfies all its contexts' constraints. Each context either scores a velocity, or marks as excluded (as it would cause a collision).

There are currently 5 contexts that are used:

- **Path following** - this is the simplest context that prefers velocities close to the path velocity.
- **Dynamic obstacle avoidance** - this context will add a high cost to any velocity that would make the agent collide with a dynamic obstacle within a specified avoidance timescale.
- **Static geometry avoidance** - whereas the original path will be guaranteed to avoid geometry, other velocities chosen by context steering could result in collisions. This context therefore excludes velocities that will cause a collision with static geometry within the avoidance timescale.

- **Repulsion** - this context provides a soft constraint to keep agents apart. If needed agents will get closer together but they will prefer to stay specified distance apart.
- **Cohesion** - this context prefers velocities that keep agents together in their group. Cohesion by default is zero, and should only be used for agents in the same group, moving together.

Agents use a gradient descent method to select the best velocity out of all available (non-excluded) velocities.

Configuration

Contextual steering can be enabled for an agent via setting the **Avoidance Mode** property on the Mercuna Ground Navigation component to **Context Steering**.

Agent-Agent Parameters

Because how an agent reacts is different for each agent it encounters, for example it might want to stay close to allies and far away from enemies, context steering uses a different set of parameters for each agent pair. These parameters are obtained through a callback to game code the first time each pair of agents interact and then are cached.

Delegate

The repulsion and cohesion parameters for other agents are read into the Context Steering system from game code using the **OnGetContextualSteeringParams** delegate on the Ground Navigation Component. This must be registered from C++, it is not exposed to Blueprint as performance of this delegate is critical.

The delegate should fill in an FMercunaContextualSteeringActorParameters struct which determines how this actor should react to the other actor (supplied as a parameter).

- **RepulsionWeight** (default 1.0) - How strong the repulsion force is, 0 indicates unused.
- **RepulsionDistance** (default 500.0) - The distance to the other agent at which the repulsion force starts being applied. The repulsion force gets stronger as the agents get closer together.
- **CohesionWeight** (default 0.0) - How strong the cohesion force is, 0 indicates unused.
- **CohesionDistance** (default 1000.0) - The distance from the other agent at which the cohesion force starts being applied. The cohesion force gets stronger as the agents get further apart. When both forces are used, this must be greater than the repulsionDistance.

Sensible ranges for the weights are 0.5 - 2.0, or 0 to disable.

There is a hard-coded maximum search multiple of 100 entity radii for considering neighbours.

Cache

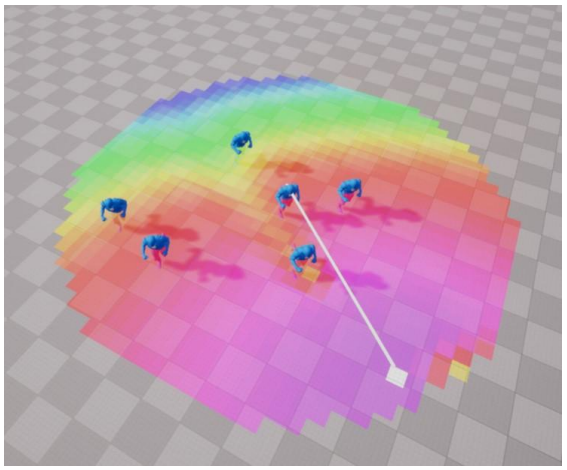
To avoid making excessive callbacks to the delegate, after the first encounter the parameters are cached on each agent. If you want to change the parameters, then you will need to invalidate the cache. The next time the agent's encounter each other they will call the delegate again. The invalidation can be done using the functions on the agent's Ground Navigation Component:

- **InvalidateContextualSteeringParams** - invalidate all cached parameters.
- **InvalidateContextualSteeringParamsAgainstActor** - only invalidate the parameters for the specified actor

Debug Draw

In order to understand how the agent's velocity is being chosen it is possible to draw the scored velocity field. In the main Mercuna menu, set a debug actor and turn on "Avoidance".

The values of a sampled velocity field are shown coloured by value (from blue=low, to magenta=high, and white=highest). Excluded velocities (i.e. that would collide with walls) are grey-ed out.



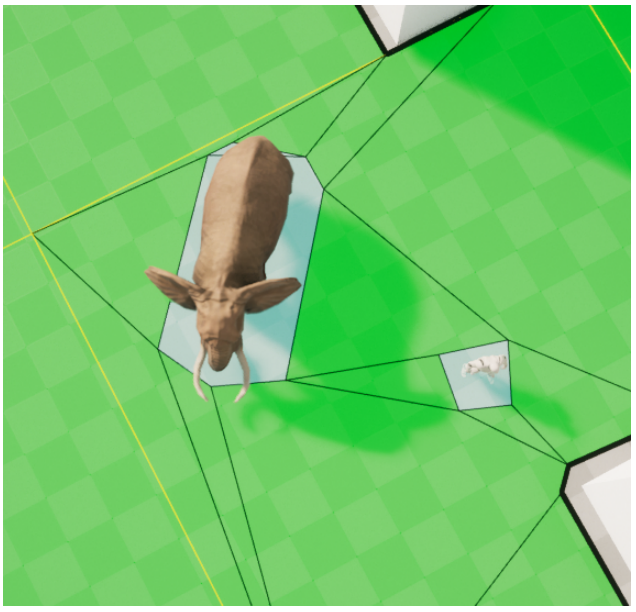
Drawing the velocity requires a regular grid of velocities to be sampled, compared with the usual method of determining the optimum velocity via gradient descent. This sampling of a very large number of velocities can have a notable impact on performance when the debug draw is active.

The white-line indicates the velocity found by a gradient descent algorithm, while the white square is found by the sampling. The latter is only performed to generate this debug draw and may not always match the white line.

Stationary Obstacles

When an actor with a Mercuna Obstacle component has stopped moving for more than a short amount of time, Mercuna has the option of cutting them out of the nav grid. Obstacles that have been cut out will be pathed around instead of being treated as a dynamic obstacle.

All paths that go near or through the obstacle that has been cut out will be automatically updated. The advantage of handling stationary obstacles through pathing, rather than dynamic avoidance means that other agents are more reliably able to move around them and won't get caught in local minima.



The **Cut Out Stationary Obstacles** property on the Mercuna Nav Grid controls whether any stationary obstacles can be cut out of that grid.

The time before an obstacle is cut out is controlled by the **Stationary Threshold Time** on the Mercuna Obstacle component. As soon as the obstacle starts moving it will be removed from the nav grid and once again treated as a dynamic obstacle.

Blueprint Functionality

The following functions are available on the **Mercuna Ground Navigation Component** and can be used to direct an agent to move between goals:

- **MoveToLocation** - move to a position, stopping within end distance of the goal.
- **MoveToLocations** - move through a series of positions, visiting them in order.

- **AddDestinationLocation** - add an additional destination to the end of the current path. The agent must currently be moving to a position or series of positions.
- **MoveToActor** - move to within a given end distance of a destination actor, if the destination moves while the pawn is moving, the path will be updated to track the destination.
- **TrackActor** - follow a given actor attempting to stay with a specified radius. If the target actor is moving the pawn will match it's speed when within the track radius.
- **CancelMovement** - immediately terminates the agent's current movement action
- **ConfigureMovement** - prompt the actor to reconfigure itself, for example if it has been moved between nav grids or the vehicle configuration has changed. Any current move actions are cancelled.
- **OnMoveCompleted** - a delegate that is triggered whenever a movement action is complete. Returns the result of the movement action as to whether it completed successfully, failed, was cancelled or was invalid (usually due to an invalid destination).
- **GetNavGrid** - Get the current nav grid this pawn is using.
- **GetTrajectory** - Returns an Unreal 5 FTrajectorySampleRange at the given sample rate, suitable for motion matching.

EQS

Mercuna currently offers four simple EQS tests. These first three of these tests are filters returning whether a point passes or fails, and do not score the points

- **Navigable** - test whether a point is within the seeded, navigable area for a pawn. Be aware that the points might be disconnected from the querying pawn, however this test is much cheaper than a reachability test.
- **Reachable** - test whether a point is reachable by a pawn from its position within a specified path distance. If the path distance is set to 0.0, then a faster test is used that only filters out points that are not connected to the pawn at all. This test does not consider the turn rate limits of the pawn, so it only guarantees that a path can be found to the target for pawns that can turn in place. If a positive ClampDistance is set, then the points will be clamped to navigable areas within this horizontal distance before testing. For generated points these will be moved to the clamped locations.
- **Raycast** - test whether there is a clear straight line path from the context to the positions.

The fourth available EQS test is:

- **Path Length** - test whether the points are reachable, and score the reachable points based on the path length from the querier to the point. This test does not consider the turn rate limits of the pawn, so it only guarantees that a path can be found to the target for pawns that can turn in place. If a positive ClampDistance is set, then the points will be clamped to navigable areas within this horizontal distance before testing. For generated points these will be moved to the clamped locations.

Additionally, Mercuna offers one EQS test that modifies the positions of the test points:

- **Project** - test whether a point is in a navigable area and project it vertically on to the nav grid. The test fails and the point is not moved if the point is further than the maximum projection distance from the nav grid.

Mercuna also offers an EQS point generator:

- **Random On Nav Grid** - Generates a number of randomly placed points on the Nav Grid generated using a uniform distribution, such that all points are within a specified path distance of the context's querying actor.

- **Grid on Nav Grid** - Generates a grid of points at navigable locations on the Nav Grid, such that they are within a given radius or path distance of the context's querying actor.

Debugging Problems

If you find that your pawn is not moving as expected, or at all, there are several debugging mechanisms available within Mercuna to help quickly identify problems.

Logging

Mercuna makes logs to the Unreal logging system to indicate progress and error conditions. By default, only Warning and Error logs are written, to enable progress information add the following to DefaultEngine.ini:

```
[Core.Log]
LogMercuna=Log
```

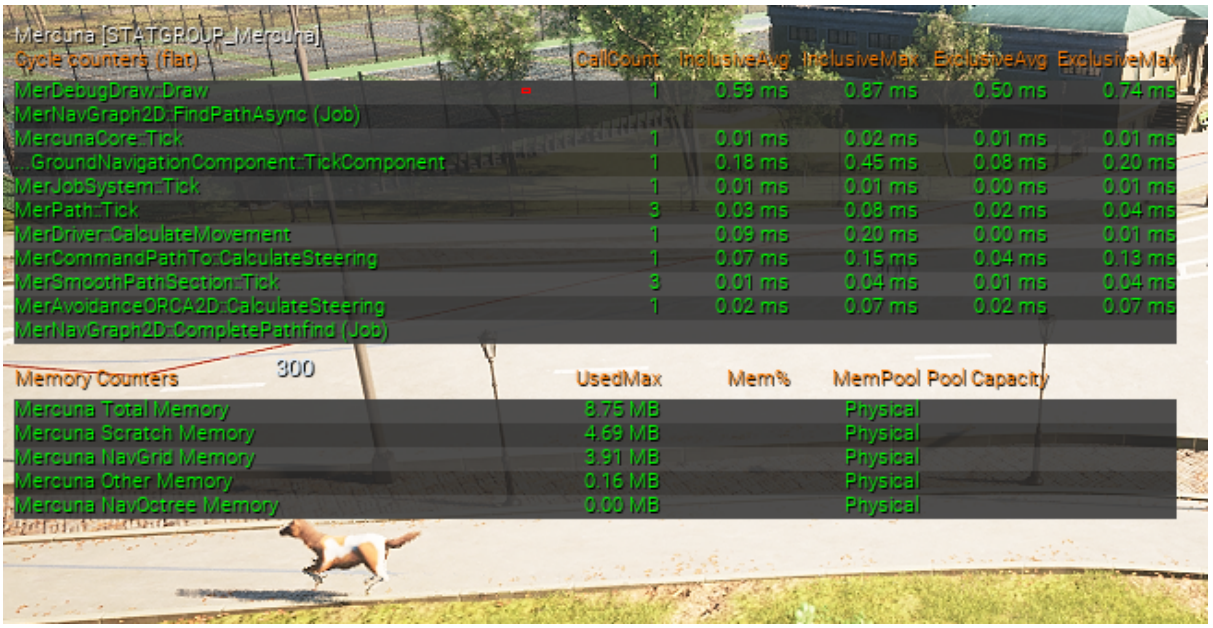
If more information is needed then Editor Preferences > Mercuna > **Enable Extra Logging**, can be enabled. This option persists between editor restarts. Extra Logging causes Mercuna to output all log messages, including additional debug messages, to a dedicated **Mercuna.log** file, located in the same directory as the standard Unreal logs.

To enable the **Mercuna.log** on standalone builds, add the following line to the Engine/Config/ConsoleVariables.ini:

```
mer.LogToFile=1
```

Profiling

Mercuna is integrated with Unreal Engine's inbuilt profiler. The current amount of time Mercuna is taking to run each frame, as well as the current memory usage, can be seen using the **stat Mercuna** console command.



Example Mercuna in editor profile

If an entry in the profiling list has *(Job)* after it's name, then it is being run as an asynchronous job on a background thread and so will normally have no effect on the frame rate.

Mercuna is also integrated with Unreal Insights and a finer grained performance profile is available through Unreal Insights.

Memory Usage

All Mercuna memory allocations are tracked and the total memory used by different systems in Mercuna is displayed through **stat Mercuna**. Note that the Scratch memory usage is a fixed amount per concurrent thread using Mercuna.

Debug Actor

When trying to understand the actions of a particular agent, it can be useful to set it as the Mercuna debug actor. This can be done by selecting the agent and setting it as the Mercuna debug actor in the Mercuna toolbar menu. The same menu also allows the debug actor to be cleared.

On screen log messages will be displayed for the Mercuna debug actor and additional debug draw is available.

In Debug builds or if the define MER_DEBUGGING is set, then additional debug logs will also be recorded to Mercuna.log for the debug actor. If you report a movement problem to Mercuna support it is helpful to include these logs and a video capture of the problem.

Actor Debug Draw

In order to help understand the current movement of Mercuna controlled actors the following debug draw is available from the Mercuna editor menu:

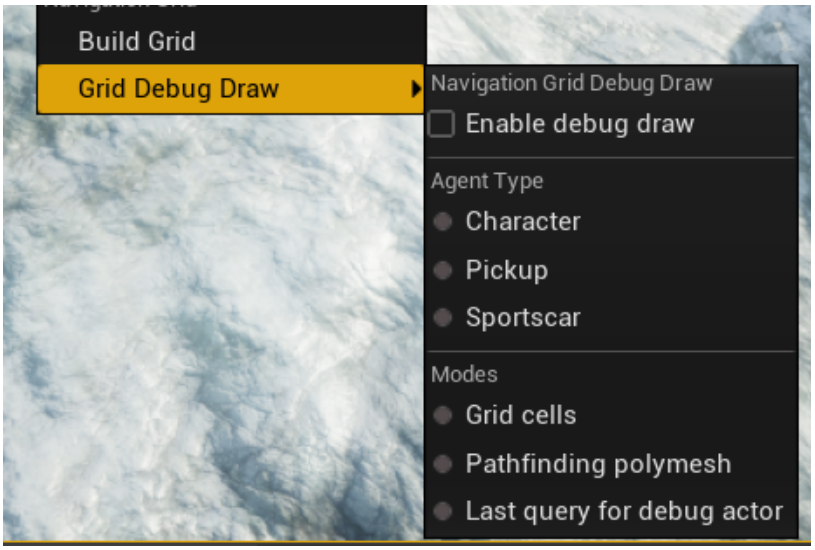
- **General**: Shows speed and velocity vector for all Mercuna actors
- **Obstacle bounds**: Shows blue spheres representing the dynamic obstacles registered with Mercuna
- **Paths**: Shows all paths currently being followed by Mercuna actors
- **Steering**: Shows the desired velocity vector for the current debug actor
- **Avoidance**: Shows various pieces of avoidance related debug draw for the current debug actor including the velocity obstacle cones and the ORCA planes, or the velocity field for context steering.

The actor debug draw can also be viewed through the Gameplay Debugger. This debug draw is replicated in multiplayer setups, allowing the Mercuna state from the server to be viewed by a client.

Navigation Grid

Debug Draw

In order to help understand Mercuna's representation of the geometry for navigation, you can draw the navigation grid, there are the following modes:



If you only have a single **Agent type** defined in your project, the navigation properties for this will be automatically displayed, otherwise you will need to select the desired agent type.

- **Grid cells** - Which cells in the navigation grid an agent could traverse (in green).
- **Pathfinding polymesh** - The polygon mesh used for accelerated character pathfinding and reachability tests.
- **Last pathfind for debug actor** - Polymesh representation (for characters) or Grid representation (for animals and vehicles) of the last pathfind for the selected debug actor or a Nav Grid Testing Actor showing A* costs.

When one of the debug draw modes is active, further advanced options are shown at the bottom of the Navigation Debug Draw Menu that allow you to further customize what is drawn.

Exporting the grid

When building a nav grid on a server, due to the lack of rendering, it can be difficult to check that the grid has been generated correctly. Mercuna therefore allows you to export the grid to an OBJ model file, that can be loaded in any common modelling program (including the Windows inbuilt 3D Viewer), via either the **SaveToObjFile** Blueprint node or the console command:

`mer.Grid.SaveOBJ` *`GridActorName AgentTypeName`* `[Polymesh]`

The exported obj file will be saved in the project root directory and named *`GridActorName`*`.obj`.

Migration Issues

v2.7

The start and end position arguments to the Nav Link Traversal callback are now relative to the Nav Grid actor transform.

v2.6

Due to changes in the way Agent Types are configured, migration from v2.6 back to v2.5 is not supported. If you upgrade to v2.6 and then downgrade back to v2.5 your Agent Type settings will be lost.

v2.5

The default Mercuna Nav Grid cell size has been changed from 100 to 20, as 20 is a more suitable default for characters. If you have any grids using the default cell size of 100 then these will automatically be changed to use cell size 20 on upgrade. Please check your nav grids if using cell size 100 intentionally (e.g. for vehicle nav grids).

v2.4

It used to be possible to turn any actor into a Mercuna Nav Seed by adding a Mercuna Nav Seed component. We have retired the seed component, as this allowed some significant performance improvements. Now only Mercuna Nav Seed actors seed the nav grid.

Known Issues

Known issues in Mercuna Ground Navigation:

- If a pawn is very close to the start of a nav link when it begins moving, it may not reach the correct orientation before entering the nav link.
- When generating the navigable area, the agent step height is added to the agent's height clearance. This ensures that the agent can safely go up any steps, but is restrictive when the agent wants to navigate under low obstacles. If your level design metrics guarantee there are no steps under low obstacles, there is a global setting that removes this padding, in Project Settings -> Plugins -> Mercuna -> Nav Grid.