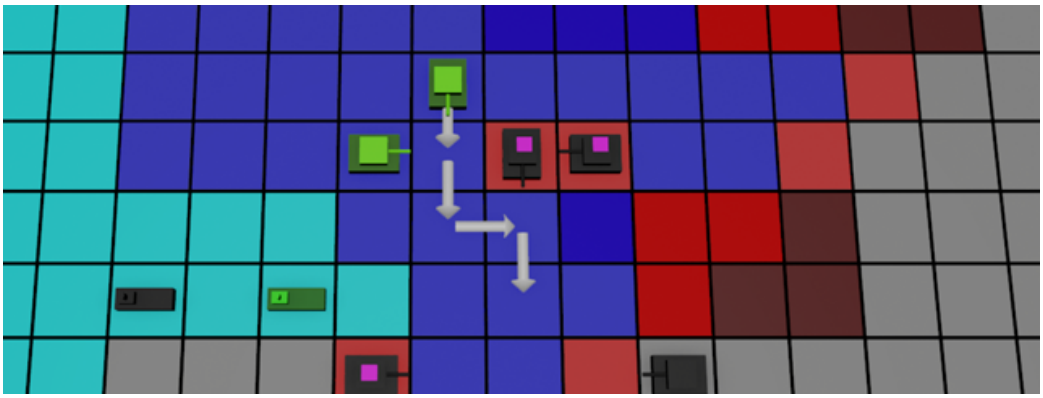
Turn Based Strategy Grids - User Guide



This version of the user guide was released on February 22, 2015. Please check the Marketplace description of the plugin regularly for updated versions.

Contents

1 Introduction **3**

2 Features **4**
- 2.1 Sparse square and hex grids 4
- 2.2 Path and attack range exploration 4
- 2.3 Targeting information 5
- 2.4 Single tile queries and search function 6

3 Blueprint usage **7**
- 3.1 IGridActor interface 7
- 3.2 LevelGrid creation 9
- 3.3 LevelGrid actor management 11
- 3.4 ActionGrid computation 12
- 3.5 ActionGrid display 15
- 3.6 ActionGrid information retrieval 16
- 3.7 Merging ActionGrids 18

4 Use cases **20**

4.1 Player input: movement commands 20
4.2 Player input: attack commands 20
4.3 Display: Action grids 21
4.4 Display: Combat forecasts 21
4.5 Display: Enemy attack range 21
4.6 AI: Simple aggressive AI 21
4.7 AI: Cost-driven AI 22
4.8 AI: Determining safe tiles 22
4.9 Use case not listed? 22

5 Miscellaneous topics **23**
5.1 How do I enable hexagon grids? 23
5.2 What coordinate system is used in hexagon grids? 23
5.3 How do I add more terrain types? 23
5.4 How do I add custom data to tiles? 23
5.5 How do I convert 2D grid ⇔ 3D world coordinates? 24
5.6 How do I detect which tile the player's cursor is hovering over? . 24
5.7 Can I enable a unit to move through others? 24
5.8 Can I add trap tiles that trigger in the middle of movement? . . . 24
5.9 Can I create a unit that occupies more than one tile? 25
5.10 Can I work with height differences? 25

1 Introduction

This is the user guide for the Turn Based Strategy Grids blueprint plugin for Unreal Engine 4. The plugin is created for the purpose of assisting developers of turn based strategy games through powerful blueprints to manage and query 2D grid worlds. Square and hexagonal grids are supported.

This is not an all-in-one turn based strategy game framework, but instead this plugin focuses on the grid management and information retrieval aspect. The plugin's intended role in your project is to, given a state of the level, compute all pathing and attack options that are available to a unit or multiple units and make it easy for you to retrieve that information in blueprint.

A high level representation of a turn based strategy game loop could look like this:

1. **Update the level layout**
2. Select a unit
3. **Determine all valid moves**
4. Select a valid move
5. Execute the move
6. Go to step 1

In this context, I use 'move' to refer to both movement and action. This plugin provides tools that solve steps 1 and 3, but intentionally leaves the remaining steps to you. I hope it will serve you well in this regard. If you have questions or feedback, do not hesitate to contact me[1] or leave a message on the plugin's forum thread[2]. Please also check the plugin's forum thread regularly for information about incoming updates.

The rest of this document is structured as follows. Section 2 lists the features of this plugin. Section 3 serves to get you started with using the included grid tools.

You can check out the interactive demo included with the plugin by importing the plugin into a project and opening the map 'TBStrategyGrids/Maps/ExampleMap'.

[1] I can be contacted at zhikang.shao@gmail.com

[2] `https://forums.unrealengine.com/showthread.php?58537-Turn-Based-Strategy-Grids-Released-Many-images`

2 Features

This section briefly lists the features of this plugin. Usage details are provided in Section 3.

2.1 Sparse square and hex grids

This plugin supports square tiled and hexagon tiled grids. It also supports sparse grids, meaning you can efficiently represent irregular shaped grids by combining smaller, rectangular shaped grids.

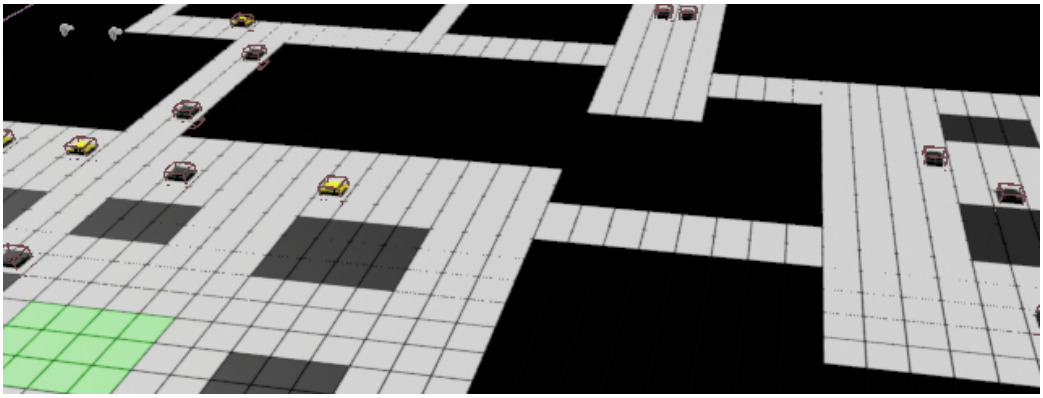


Figure 1: Irregular shaped grids can be efficiently represented by composition using smaller, rectangular shaped grids with either square or hex tiles.

2.2 Path and attack range exploration

Compute a unit's pathing and attack options for one turn. Paths can be retrieved to any tile within movement range. Supports multiple terrain types that affect movement and targeting. Some units may move through others. Unit-terrain and unit-unit interaction can differ per unit and is customizable. Line of sight check can be enabled when searching for tiles within attack range.

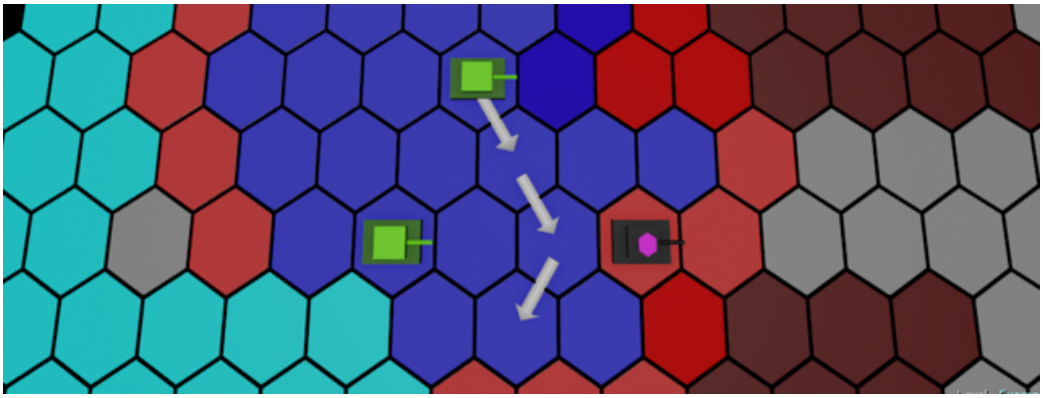


Figure 2: This plugin computes a unit's pathing and attack options for you. Supports optional line of sight check, different terrain types and some units being able to pass through others.

2.3 Targeting information

In addition to storing which tiles are in attack range, computed grids can also store per tile which (actor) targets are within range of that tile, meaning 'from tile X, I can attack target Y'. Separate lists are used for different allegiances to the acting unit, where allegiances can be neutral, friendly or hostile.

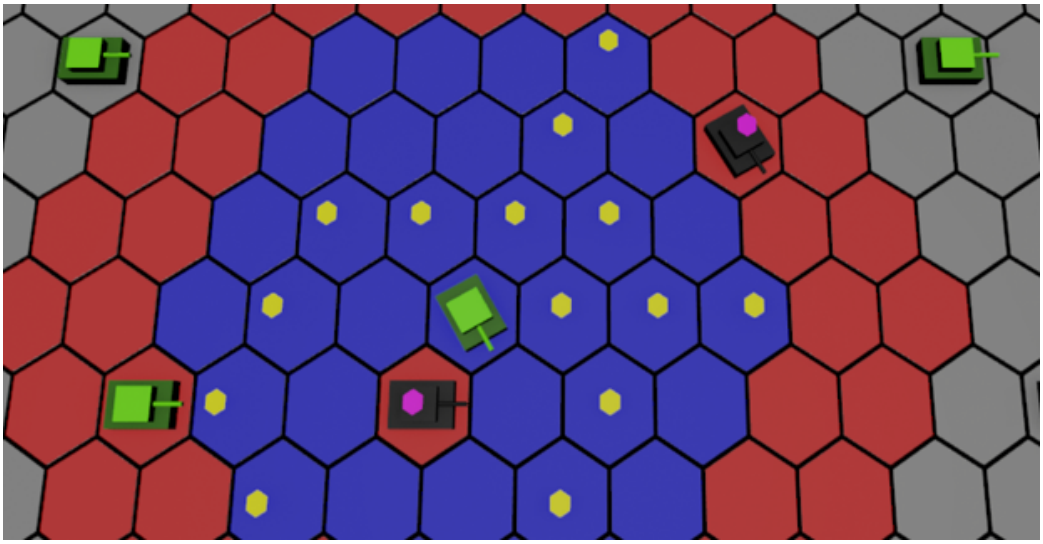
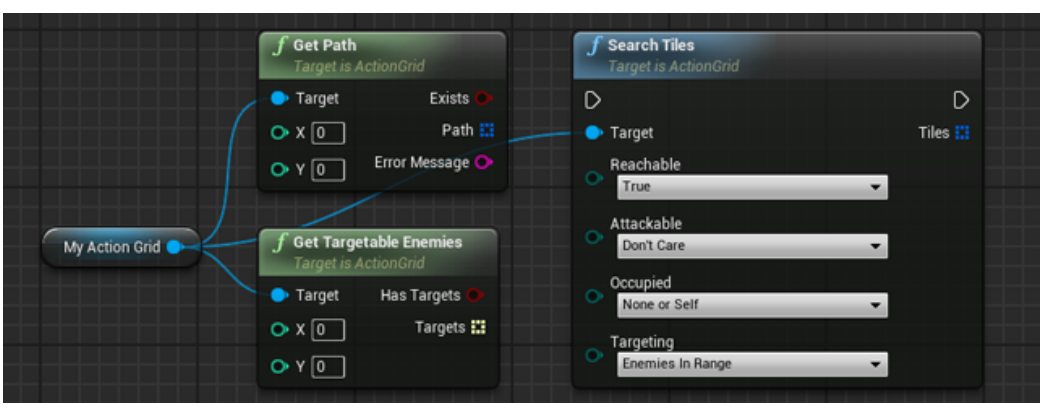


Figure 3: An advanced optional feature is to store at each tile the units that are within attack range. In this image the acting unit has min-max attack range 2-2 and resulting tiles from which he can attack enemies are marked yellow.

2.4 Single tile queries and search function

Information can be retrieved for a specific tile or you can use a search function that returns a list of tiles. Generally single tile queries will be helpful when validating and processing player input and scripted AI, while tile lists are useful for display and high level AI.



3 Blueprint usage

This section will get you started with using this plugin. The three assets in this plugin that you will be interacting with mainly are: the LevelGrid blueprint, the ActionGrid blueprint and the IGridActor interface.

- A **LevelGrid** represents the state of your level, specifically: the placement of units and the terrain type of each tile.
- **ActionGrids** store the movement and attack options available to one or multiple units given the current state of the level
- Any blueprint class that implements **IGridActor** represents a unit that can interact with LevelGrids and ActionGrids

Furthermore, the **GridUtil** blueprint library contains useful utility functions, such as 3D world ⇔ 2D grid transformations (both square and hex) and a tile picking function to determine which tile the player's cursor is currently hovering over. The functions in GridUtil are globally available.

3.1 IGridActor interface

To make your character blueprint class interactable with this plugin, make it implement the IGridActor blueprint interface. Open your character blueprint and open up its properties by clicking the Blueprint Props button. In the details panel find the Interfaces category. Select Add ⇒ IGridActor. Your character blueprint now implements the interface, but you must still make the interface's functions behave like intended before they can be used with the grids.

Tip: The units in the interactive demo are of the ExampleCharacter blueprint class. You can use this as a reference when implementing the interface.

Go to Graph Editing Mode and implement the interface functions as described below. Functions with return values appear in the 'My Blueprint' tab under 'Interfaces', so open their graphs to start implementing their functions. Functions without return values, that is SetGridPosition and SetGridTransform, don't appear here but can be implemented in the general event graph. Right click anywhere in the event graph and create Event Set Grid Position and Event Set Grid Transform nodes and implement the function from there.

IGridActor function	Description
SetGridPosition	Notifies unit of his position on the grid
GetGridPosition	Gets the unit's position on the grid
SetGridTransform	Sets the grid transform to be used by this unit
GetGridTransform	Gets the grid transform used by this unit
CanEnterTerrain	Define terrain accessibility and movement cost
CanTargetTerrain	Define terrain targetability
CanPassThroughActor	Define passability through other unit
GetAllegianceToActor	Define allegiance to other unit

For the Set/Get function pairs, make sure that any values that are passed when Set is called are returned the next time Get is called. These functions are called by the LevelGrid when the unit is added or moved. Do not call SetGridPosition yourself, instead place and move the actor using LevelGrid functions.

The latter four functions are used to retrieve the unit's movement and attack capabilities. I'll present their purpose, but their implementation is up to you. I use *FunctionName(Input) : Output* to list a function's input and output variables.

- **CanEnterTerrain(T) : CanEnter, MoveCost** Given a terrain type T (enum), whether this unit can enter a tile of that terrain type and if so, the movement cost for doing so. Movement cost must be positive and non-zero. Any movement cost ≤ 0 is automatically replaced by 1 in computations. If you want to get started quickly, just set CanPass to true and MoveCost to 1. *Note: The default terrain types BlockingCube and BlockingPillar are treated as exceptions and are never passed to this function, they always block.*
- **CanTargetTerrain(T) : CanAttack** Given a terrain type T, whether this unit can target tiles of this type. If the unit cannot target terrain type T, tiles of that type are never flagged as attackable in ActionGrids. Use this if you want to make the targeting capabilities of this unit influenced by terrain, otherwise just set CanAttack to true.
- **CanPassThroughActor(A) : CanPass** Whether this unit can pass through the other unit A. The interactive demo demonstrates how to decide this based on TeamID.
- **GetAllegianceToActor(A) : Allegiance** Whether this unit considers the other unit A to be an ally, enemy or neutral. The allegiance is used for storing targets in an ActionGrid, which can be searched for friendly, enemy

or neutral targets later.

After implementing the Get/Set function pairs and defining the unit's movement and attack capabilities through the other functions, your unit class is set up to be used with LevelGrids and ActionGrids.

3.2 LevelGrid creation

The shape of a LevelGrid can be defined in the editor and at run-time. Editing in the editor gives you the advantage that you can see changes to the layout of the level reflected in your viewport while editing. Being able to manipulate the LevelGrid at run-time means that you can use this plugin with procedurally generated tile based maps.

Start by dragging a LevelGrid blueprint into your world. A 20x20 grid should show up at the world origin, we will change this in a moment. With the LevelGrid selected, head over to the Details panel. Expand and edit the GridTransform properties to change the placement and tile spacing of the grid and to define whether the grid has **square** or **hexagonal** tiles.

Note: When doing WorldToGrid or GridToWorld transformations a GridTransform is required. Use the LevelGrid's GridTransform for this. Units that have been added to a LevelGrid receive a copy via a SetGridTransform call.

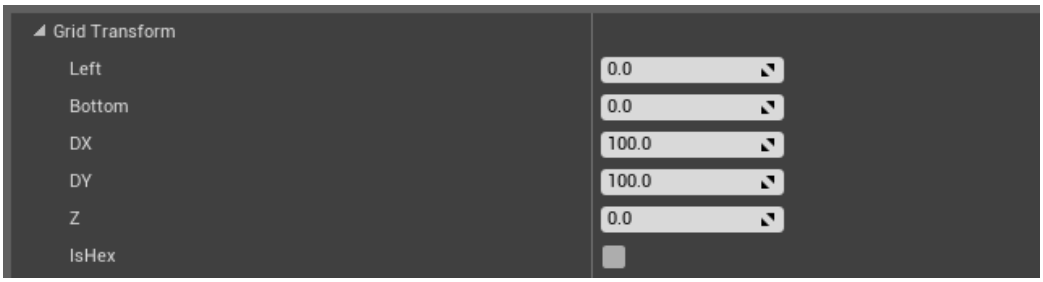


Figure 4: The LevelGrid's GridTransform defines grid placement, spacing and square/hex type. GridTransforms are also used in the GridUtil's 2D grid ⇔ 3D world conversion functions.

Now its time to define the global shape of the grid. Locate and expand the InitialRooms list. Rooms are rectangular sections with a default terrain type,

which together make the play area. By default the InitialRooms list contains one room, which is the grid you are seeing right now.

Add, remove or change rooms to change the play area. The changes are directly visible in your editor viewport. Rooms are allowed to overlap, in which case the terrain of the room that appears earlier in the InitialRooms list takes precedence. If you want, you can just create one large room and move on.

To add rooms after play has started (for procedural maps), obtain a reference to a LevelGrid and call the LevelGrid's AddRoom, SetTerrainType and RefreshTerrainPreview functions:

- **LevelGrid** ⇒ **AddRoom(X, Y, W, H, T)** Creates a room at the given rectangular bounds and initializes each tile's terrain type to T .
- **LevelGrid** ⇒ **SetTerrain(X, Y, T)** Sets the terrain type of the tile at X, Y to T.
- **LevelGrid** ⇒ **RefreshTerrainPreview()** Uses the current room and terrain data to refresh the colored tile preview. Call this when you're done making run-time modifications (for the time being). Only necessary if you use the colored tile preview.

Another list is the InitialTerrain list. You can use this to change the terrain type in rectangular regions before play has started. In contrast to InitialRooms, the values in the InitialTerrain list do not create new rooms and thus do not expand the play area. Once you're done setting up the shape of the grid and the initial terrain, its time to add actors.

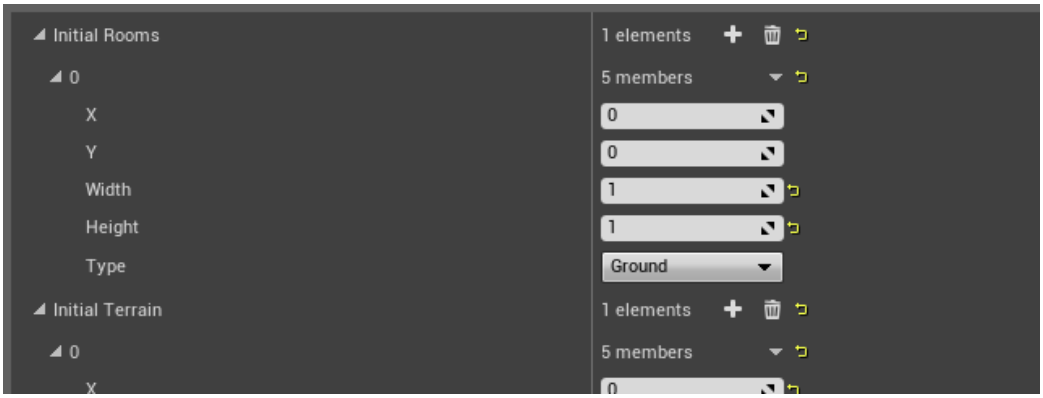


Figure 5: Use a LevelGrid instance's InitialRooms and InitialTerrain lists to specify the grid's initial shape and terrain.

3.3 LevelGrid actor management

Actors that implement the IGridActor interface can be managed on the LevelGrid using the LevelGrid's AddActor, MoveActor and RemoveActor functions:

- **LevelGrid ⇒ AddActor(Actor, X, Y)** Adds actor A to the grid at position X, Y. Be sure to only add actors that aren't currently added to another LevelGrid.
- **LevelGrid ⇒ MoveActor(Actor, X, Y)** Moves actor A that is on the grid to position X, Y. Use this when the actor has been added before.
- **LevelGrid ⇒ RemoveActor(A)** Removes actor A from the grid.

To change the grid position of a unit that has been added to the grid you should only use MoveActor. This ensures that the LevelGrid has a correct representation of the game world at all times.

Note that MoveActor instantly moves an actor from his current position to his new position. The LevelGrid only cares about states for which he potentially has to do computations. If a unit moves along a path, the LevelGrid is only interested in the position he ends at.

You can choose to add actors manually, but you can also choose to register them at game start based on their world location. Use the globally available function FindAndRegisterUnits to do this. This function takes a reference to a LevelGrid as input and iterates over all IGridActors in the world. It uses the

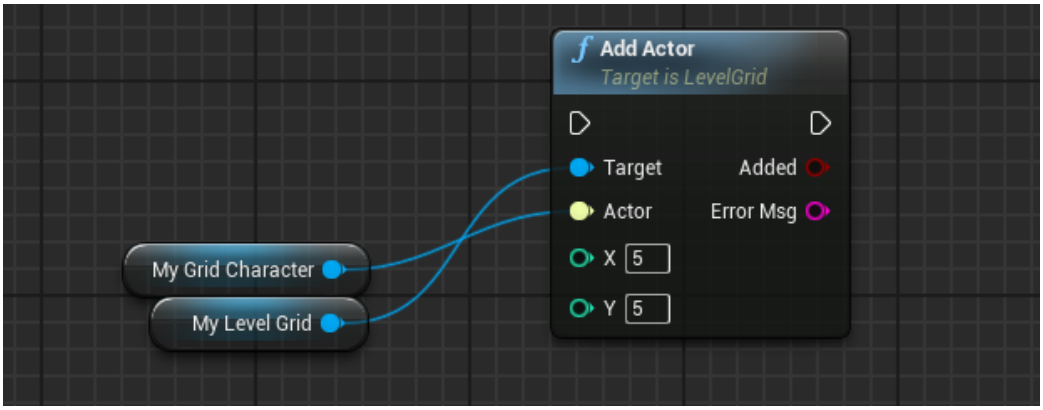


Figure 6: The blueprint node for adding an actor to a LevelGrid.

LevelGrid's GridTransform to translate each unit's 3D world position to a 2D grid position and will register it to the LevelGrid only if the computed tile is valid and unoccupied.

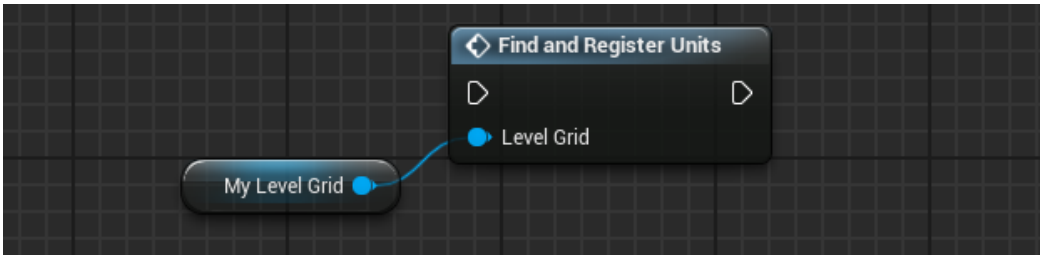


Figure 7: The blueprint node for automatically registering all actors placed in the level based on their world position.

3.4 ActionGrid computation

After you have added units to a LevelGrid, you can compute for any unit what pathing and attack options are available to him given the current state of the LevelGrid. This can be done by calling the LevelGrid's ComputeActionGrid function. It will result in an ActionGrid being created that stores all options available to the unit.

ActionGrids provide useful information for the active unit but can also be used to

provide information about idle units, for example to preview their attack range. The following options are available when calling ComputeActionGrid:

- **Max Movement Cost** The maximum movement cost to compute with
- **Min-Max Attack Range** The min and max attack range to compute with
- **Line of Sight Check** Whether to do a line of sight check when determining attackable tiles
- **Compute Targets** Whether to compute and store which targets can be attacked from each tile within movement range
- **Auto Destroy** Whether to let the plugin automatically clean up the created ActionGrid once it is outdated

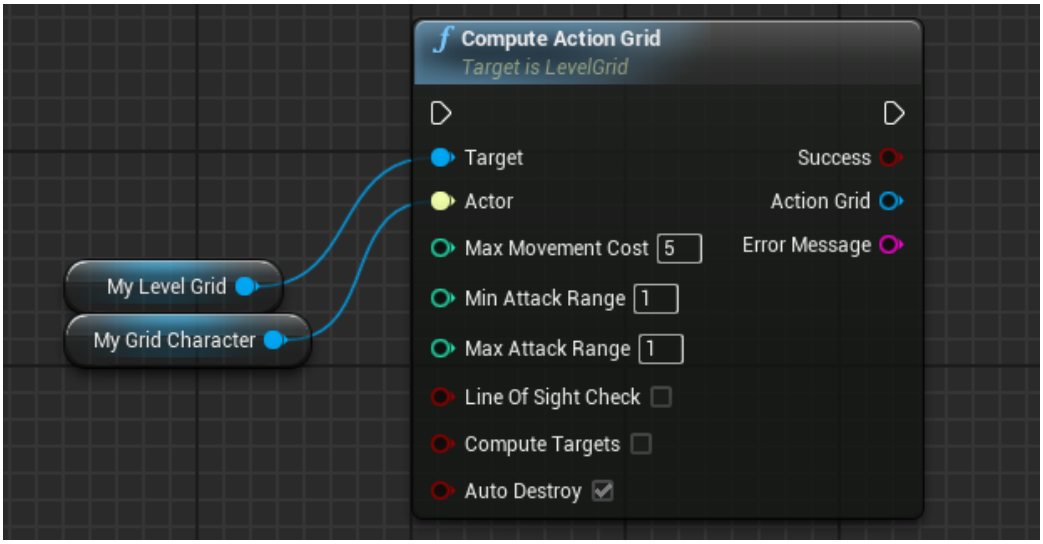


Figure 8: Use a LevelGrid instance's InitialRooms and InitialTerrain lists to specify the grid's initial shape and terrain.

Tip: The interactive demo allows you to change these parameters on the fly. Check it out to get a feel for what action grid computation does.

The ActionGrid computation first explores around the unit to find all **reachable** tiles. It flags those tiles as reachable and stores the **shortest paths** to those tiles. Then it determines from each reachable tile which tiles are in attack range and flags them as **attackable**. To only perform one task and not the other, simply set the other task's values to 0. For example, set the min-max attack range to 0-0 to only compute movement. Set the maximum movement cost to

0 to only compute attackable tiles.

Any other units encountered when exploring or determining attackable tiles are also stored in the ActionGrid, along with their allegiance (friendly/hostile/neutral) to the current unit. They are stored in two ways: any tile in the ActionGrid stores a reference to the unit occupying that tile. Units encountered on attackable tiles are also stored in the tiles they can be attacked from. The last feature is only executed when **compute targets** is checked. Leave it unchecked to save some computation time when you don't need targeting info.

A **line of sight check** can be enabled so that players can not attack through terrain types that block vision. Disable this if your game does not require it. The line of sight check traces a vision line between the centers of two tiles to determine whether the line of sight is clear. Which terrain types block vision is determined in the LevelGrid functions CanVisionPassThroughCenter and CanVisionPassThroughCorner. By default two terrain types block vision but with a subtle difference: BlockingCube tiles block vision completely, while BlockingPillar tiles block vision but not if the vision line passes exactly through the corner of the tile.

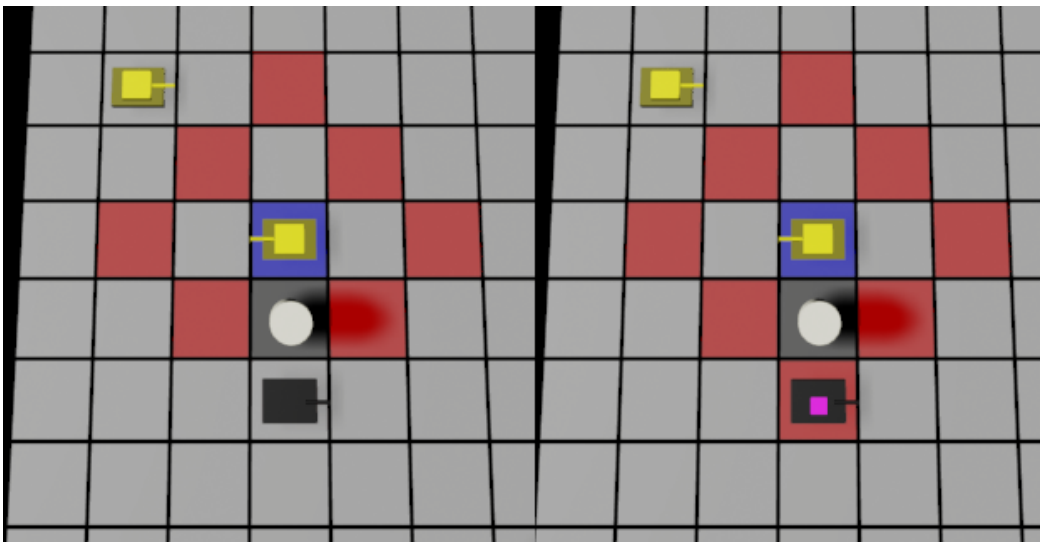


Figure 9: The result of line of sight check disabled (left) vs. enabled (right).

The **auto destroy** option specifies whether the LevelGrid should automatically destroy the created ActionGrid once it is outdated, allowing it to be garbage

collected. It is considered outdated as soon as the state of the LevelGrid changes, i.e. when a unit is added, moved or removed. If you checked auto destroy, trying to access an outdated ActionGrid will result in a warning being printed to the screen. I recommend leaving auto destroy checked; only uncheck it if you intentionally want to work with outdated ActionGrids.

3.5 ActionGrid display

The included ActionGridView blueprint exists for the purpose of displaying ActionGrids in-game. An ActionGridView instance can be reused as many times as you want, so that you do not have to spawn a new instance for every ActionGrid to display. To display an ActionGrid, you first require a reference to an ActionGridView. One way to do this is to drag an ActionGridView into the editor viewport before the game starts and find it using a GetAllActorsOfClass node.

After having obtained a reference to an ActionGridView, you can call the ShowActionGrid function to show an ActionGrid in the world. It will display reachable tiles as blue and attackable tiles as red, where blue has priority over red. Feel free to edit the representation or to make your own ActionGrid visualization class. In that case you can use ActionGridView as a reference for how to retrieve the information necessary for display from the ActionGrid. I will also discuss that next.

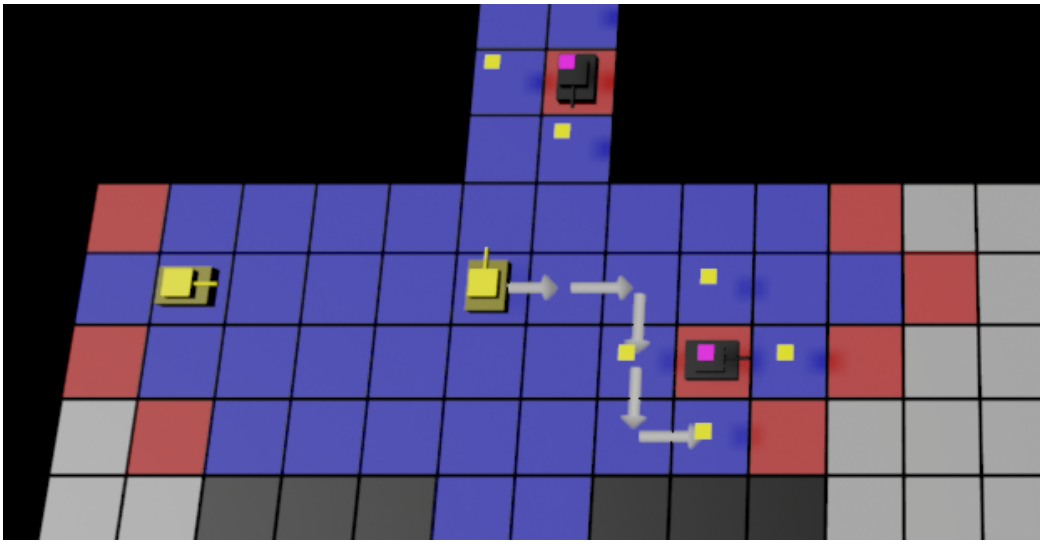


Figure 10: The ActionGridView blueprint can be used to display ActionGrids.

3.6 ActionGrid information retrieval

The following information can be retrieved from ActionGrids:

- Whether a tile is reachable and if so, a shortest path to that tile
- Whether a tile is attackable from any reachable tile
- The unit occupying a tile
- All neutral targets that can be attacked from a tile
- All friendly targets that can be attacked from a tile
- All hostile targets that can be attacked from a tile

To retrieve this information from a specific tile, call the following functions on an ActionGrid:

- **IsReachable(X, Y)** Returns whether X, Y is reachable
- **GetPath(X, Y)** Returns a shortest path to X, Y, only if reachable
- **IsAttackable(X, Y)** Returns whether X, Y is attackable
- **IsOccupied(X, Y)** Returns whether X, Y is occupied
- **GetOccupant(X, Y)** Returns the occupant and its allegiance
- **GetTargetableEnemies(X, Y)** Returns a list of targetable enemies
- **GetTargetableFriendlies(X, Y)** Returns a list of targetable friendlies
- **GetTargetableNeutrals(X, Y)** Returns a list of targetable neutrals

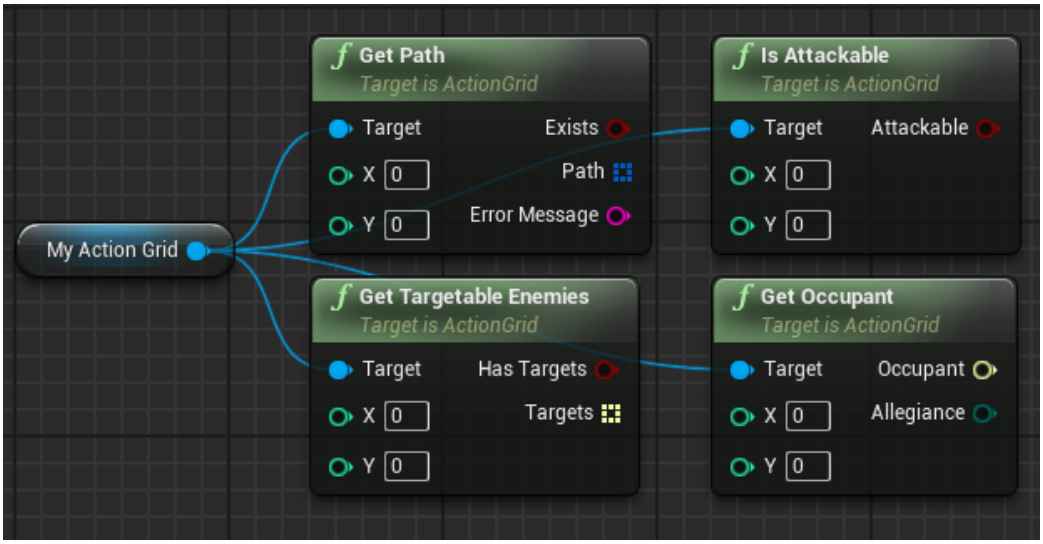


Figure 11: Some single tile data retrieval nodes.

GetPath returns a list of Step structs, which represent the steps of the path in order excluding the unit's current position. The Step struct contains an X, Y position which is the tile the step ends at. It also contains a DX, DY and a Dir (enum) value which represent the unit step taken from the previous tile to the new tile X, Y. Both DX, DY together and Dir represent the same direction. Its up to you to decide which information in a Step struct you prefer to use.

The target lists that are retrieved by calling GetTargetable... contain exactly the targets that were within the min-max attack range measured from the queried tile and only those targets that passed the line of sight check if that was enabled. To re-iterate: this feature only works if you checked **compute targets** when computing the action grid.

As an alternative to single tile queries you can search for all tiles in an ActionGrid that satisfy a number of conditions. This can be done by calling the **SearchTiles** function on an ActionGrid. You can specify one or multiple search conditions at once. The following search conditions are available:

- Reachable: don't care, true or false
- Attackable: don't care, true or false
- Occupied: don't care, unoccupied, occupied by friendly/enemy/neutral
- Targeting: don't care, has friendly/enemy/neutral targets

The SearchTiles function returns a list of tile coordinates that satisfy all conditions that you specified. You can then process those tiles further. For example, an AI healer could search for all tiles from which he can target friendly units and then process the returned list of tiles to decide a final tile to move to. Note that similar to the single tile queries, the targeting condition only works correctly if you had 'compute targets' checked when computing the action grid.

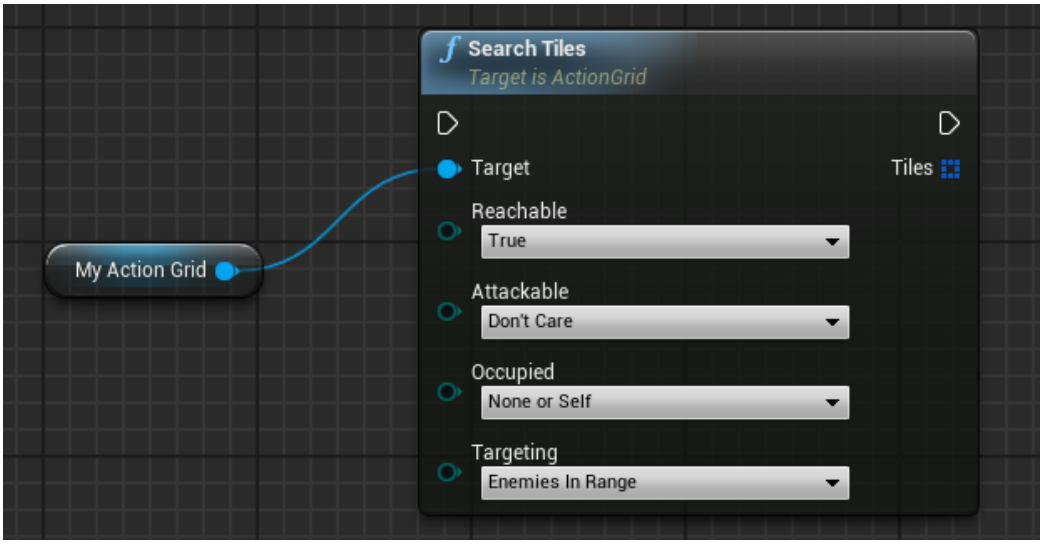


Figure 12: The ActionGrid's search function.

3.7 Merging ActionGrids

Merging ActionGrids is useful in a number of situations, primarily when wanting to know the movement and attack range of enemy units as a player or as AI. This can be done using the MergeActionGrids function defined in GridUtil. This function takes two ActionGrids and produces a new one with the combined information. All information (reachable, attackable, occupant, target lists) is retained except for paths.

The included ActionGridView blueprint has a color scheme prepared for ActionGrids that represent enemy information: call ShowActionGrid with IsEnemyPreview set to true. This color scheme has a red tint in general, similar to display of enemy info in the Fire Emblem games.

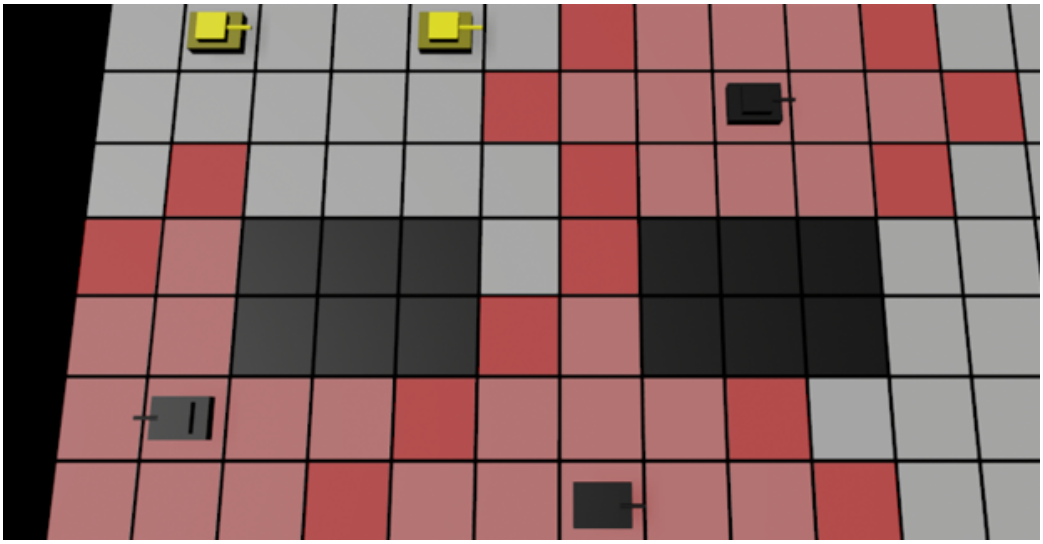


Figure 13: ActionGrids can be merged to represent the movement and attack radius of multiple units.

4 Use cases

I will now present a number of use cases for ActionGrid functions. These are meant to give you an idea of in what kind of situations ActionGrids can assist you and which functions to use in those cases. Tasks that are not provided by this plugin and are non-trivial are enclosed in square brackets [ ].

4.1 Player input: movement commands

It's the player's turn. We compute an ActionGrid with MaxMovementCost > 0 for the player's selected unit and display it on screen using an ActionGridView. As the player moves his cursor, we use GetPath(X, Y) to check whether a path exists and if so, preview the path using the same ActionGridView.

The player clicks a tile. We compute which tile X, Y was clicked using the PickTile function defined in GridUtil. We call IsReachable(X, Y) on the ActionGrid to see whether the clicked tile is reachable. It is, so we call GetPath(X, Y) to retrieve the path and animate the unit along the path.[3] *Note: This use case is implemented by ExampleController.*

4.2 Player input: attack commands

It's the player's turn, but he has spent his movement so we compute an ActionGrid with MaxMovementCost $= 0$. The player clicks an enemy unit. We determine what tile X, Y the unit is standing on by casting it to an IGridActor and calling GetGridPosition. We call IsAttackable(X, Y) on the ActionGrid to see if the enemy's position is within attack range. It is, so we [start combat] between the two units.

Alternatively, the player clicks a tile X, Y instead of clicking a unit directly. We do the same checks.

[3] You can use the included PathAnim blueprint to slide and rotate an actor along a path retrieved from an ActionGrid. Implement IPathAnimListener and call SetListener to receive an event once the path animation completes.

4.3 Display: Action grids

It's the player's turn. We compute an ActionGrid to determine the tiles he can reach and the tiles he can attack (movement included). We use the search function to find all tiles that are reachable and spawn a temporary blue overlay for those tiles. We use the search function again to find all tiles that are attackable but not reachable and spawn a red overlay for those tiles. *Note: This use case is implemented by ActionGridView.*

4.4 Display: Combat forecasts

It's the player's turn and his movement is spent so we compute an ActionGrid with MaxMovementCost = 0. We call GetTargetableEnemies(X, Y) on the player's current position to get all enemies within attack range. We use this list to [show combat forecasts] above each enemy.

4.5 Display: Enemy attack range

It's the player's turn. The player wants to view the movement and attack range of a set of enemy units. We call ComputeActionGrid on each individual unit. Then, we call MergeActionGrids to merge them and use an ActionGridView to display the result.

4.6 AI: Simple aggressive AI

It's the AI's turn. We compute an ActionGrid including both movement and attack range. We **search** for tiles with enemy targets within attack range. We select a random tile returned from the list and move to that tile. We remain idle of no such tile was found. After moving, we call GetTargetableEnemies on the AI's new position and [attack] a random enemy from the list.

Support AI can be done similarly by moving to friendly targets and [using support abilities] on friendly targets.

4.7 AI: Cost-driven AI

Assumes you have defined a function $Cost(X, Y, ...)$ that returns how preferable a tile position is. We first **search** for tiles filtering out any unwanted tiles (for example: tiles that are currently occupied and tiles that have no enemy targets). The remaining tiles are assigned a score using $Cost(X, Y, ...)$ and the tile that scores highest is selected to move to. Optionally have a back-up strategy if the first strategy yields no results.

4.8 AI: Determining safe tiles

It's the AI's turn. He would like to move to a position where he is safe during the enemy's next turn. We compute an ActionGrid for each individual enemy unit and call MergeActionGrids to merge them all together. We can then call SearchTiles with condition Attackable: False to retrieve a list of all tiles that are outside the enemies' attack range or call IsAttackable for a specific tile.

4.9 Use case not listed?

If you have a use case that involves grid computations that is difficult to do in the plugin as it is now, please send your feedback to me (see Section 1) so I can consider it for future releases.

5 Miscellaneous topics

5.1 How do I enable hexagon grids?

Both grid types are available through the same LevelGrid class. Open a LevelGrid instance' Details, expand GridTransform and then check/uncheck IsHex. All internal computations will adapt accordingly.

5.2 What coordinate system is used in hexagon grids?

Please see Amit Patel's (RedBlobGames) article[4] on hexagons for terminology. If you're using hexagon grids, all public LevelGrid and ActionGrid functions expect offset coordinates. Axial coordinates are used internally for path exploration in LevelGrid and path retrieval in ActionGrid.

5.3 How do I add more terrain types?

To define more terrain types, add entries to the ETerrainType enum. Then update your unit class' implementation of the IGridActor functions CanEnterTerrain and CanTargetTerrain to define how your units interact with the new terrain types.

If you're using the LevelGrid's colored tile visualization (ShowTiles is checked), update the LevelGrid's TerrainTypeToColor function. If tiles of your new terrain type should block vision, update the LevelGrid's CanVisionPassThroughCenter and CanVisionPassThroughCorner functions.

5.4 How do I add custom data to tiles?

Data that is part of the level (and not a unit's turn) can be added by adding a variable to the LevelTile struct. You can then create Get/Set functions for that variable by mimicking the Get/SetTerrain and Get/SetActor functions. Please update the existing Get/Set functions for level data so that no data is lost when

[4] `http://www.redblobgames.com/grids/hexagons/`

modifying structs. This is currently a limitation of Unreal Engine when working with structs.

5.5 How do I convert 2D grid ⇔ 3D world coordinates?

These transformations can be done using the globally available GridToWorld and WorldToGrid functions. They require a GridTransform which describes the grid's origin, tile spacing and square/hex type. You can obtain one from a LevelGrid or any unit that has been added to a LevelGrid.

5.6 How do I detect which tile the player's cursor is hovering over?

Use the globally available PickTile function for this. This function requires a PlayerController reference (for the cursor) and a GridTransform reference (for the grid placement and square/hex type). Use a GetPlayerController node to get the first player controller. Other player controllers can be used if dealing with split-screen multiplayer. It also requires a GridTransform, which can be obtained from a LevelGrid or any unit that has been added to a LevelGrid.

5.7 Can I enable a unit to move through others?

Yes, this depends on your unit class' implementation of the IGridActor function CanPassThroughActor. To allow unit A to move through unit B, CanPassThroughActor when called on unit A should return true when unit B is passed as parameter.

5.8 Can I add trap tiles that trigger in the middle of movement?

Yes, since animation along a path and updating the LevelGrid at the end of a path are your responsibility. You can retrieve a path from an ActionGrid then manually check each tile along the path for traps. If a trap is detected and the

unit stops his movement there, update the LevelGrid by moving the unit to the position where it was stopped.

5.9 Can I create a unit that occupies more than one tile?

At this moment units are assumed to always occupy one tile. For stationary units you can fake them occupying more tiles by setting the terrain types of surrounding tiles to a movement blocking type. No better options are available at this time.

5.10 Can I work with height differences?

At this moment path planning and attack range calculation do not take into account height differences.