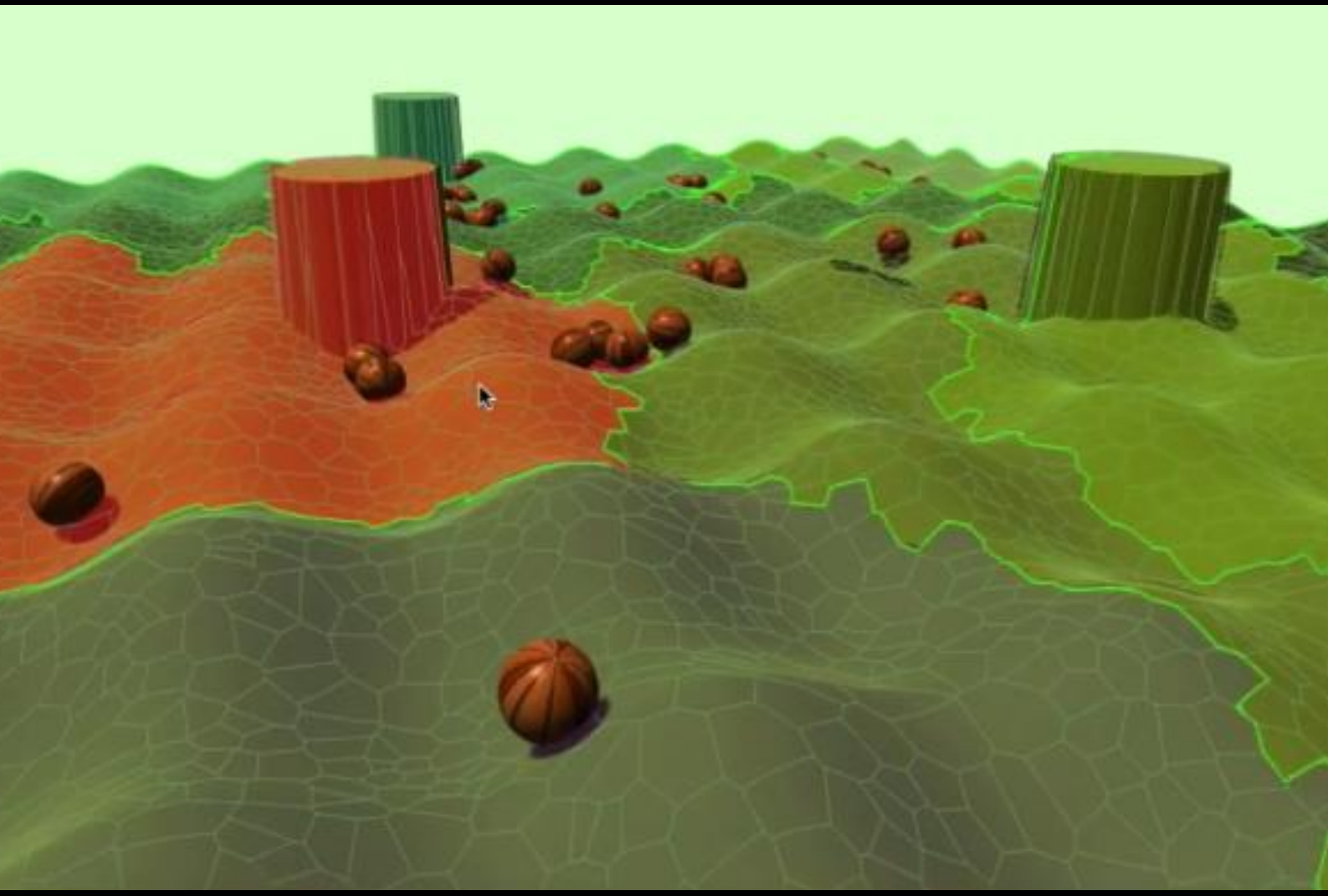
TERRAIN GRID SYSTEM



USER GUIDE

Index

Introduction **3**

QuickStart and Demo Scenes **3**

Support & Contact info **3**

How to add the grid to your scene **4**
In Unity Editor *4*
At runtime *4*

Custom Editor Properties **5**
Grid Configuration *5*
Grid Positioning *6*
Grid Appearance *7*
Grid Behaviour *8*
Path Finding *9*
Grid Editor *10*

Positioning the Grid on your terrain **11**

Changing default Input system **11**

Programming Guide (API) **12**
Use Terrain Grid System from scripting *13*
Public API structure *14*
Complete list of public properties, methods and events *15*
Basic grid functions 15
Cell related 16
Territory related 20
User interaction / Highlighting 22
Path Finding & LOS related 23
Other useful instance methods 24
Other useful static methods 24
Events 25
Handling cells and territories 25
Rectangle Selection 26
Custom user data 27

Introduction

Thank you for purchasing!

Terrain Grid System is a commercial asset for Unity 2019.4 (or later) that allows you to:

- Add a configurable and flexible grid to Unity terrain or any other gameobject.
- Supports voronoi tessellation, boxed and hexagonal types.
- Two levels of regions: cells and territories.
- Powerful selectable and highlighting system for both cells and territories.
- Coloring and fade support.
- Different positioning options for best mesh adaptation to the terrain.
- Native A* Path Finding support.
- Extensive API (C#) for controlling the grid.
- Use textures to define visibility and cell ownership
- Can work alone or with Unity's Terrain object

You can use this asset for:

- Organizing content or areas of interest over a terrain
- Provide a generic selectable grid zone without using a terrain
- Allow the user to highlight and select a zone of the map
- Display with different colors regions of the map, either territories or cells
- Manage the grow of different regions, by combining cells into greater cells or other territories.

QuickStart and Demo Scenes

1. Import the asset into your project or create an empty project.
2. Go to Demo folders and run them to learn about common use cases.
3. Examine the code behind the script attached to the Demo game object.

The Demo scenes contain a TerrainGridSystem instance (the prefab) and a Demo gameobject which has a Demo script attached which you can browse to understand how to use some of the properties of the asset from code (C#).

Support & Contact info

We hope you find the asset easy and fun to use. Feel free to contact us for any enquiry.
Visit our Support Forum for usage tips and access to the latest beta releases.

Kronnect
Email: contact@kronnect.com
Kronnect Support Forum: https://www.kronnect.com/support

How to add the grid to your scene

In Unity Editor

There're 3 ways to add Terrain Grid System to the scene in Editor time:

1) If you're using Terrain Grid System for 2D (not terrain based) select the menu GameObject -> 3D Object -> Terrain Grid System and the grid will be added to the scene.
2) If you wish to add a grid to a Terrain or other gameobjects like mesh-based terrains, just select your terrain or root of gameobjects and add the script "Terrain Grid System". It will configure automatically for you.
3) Alternatively, you can drag the prefab "TerrainGridSystem" from "Resources/Prefabs" folder to your scene and assign your terrain or gameobject to the Terrain property.

Once created, select the GameObject created to show custom properties and assign the terrain gameobject reference.

At runtime

You can also add Terrain Grid System dynamically in Play Mode in 2 ways:

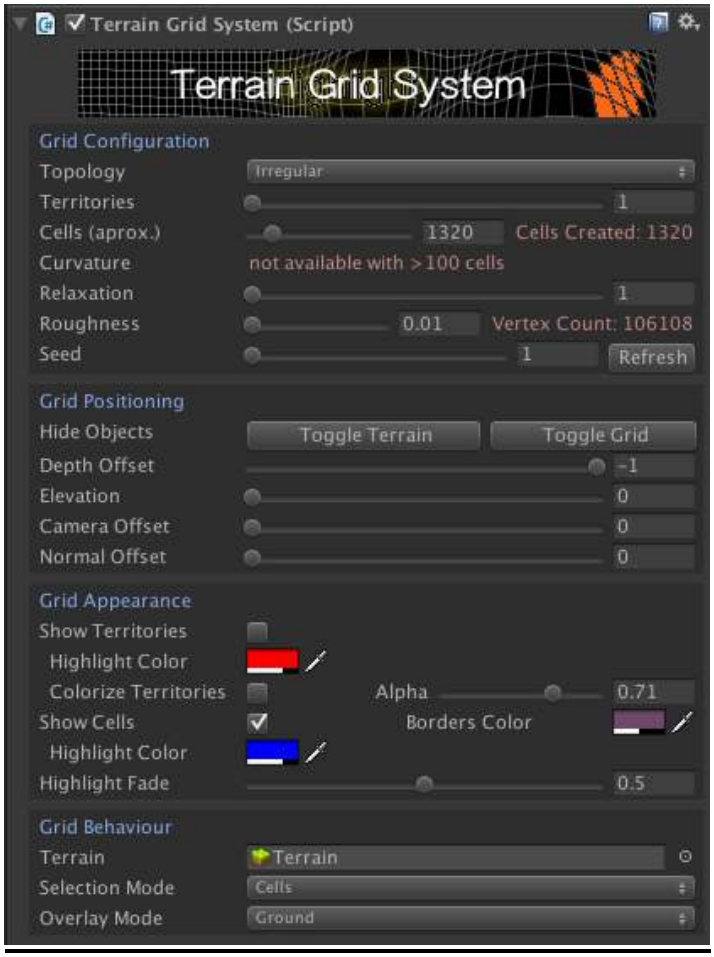
1) Call AddTerrainGridSystem() method on your terrain or gameobject. Example:

   ```
   TerrainGridSystem tgs = myTerrainGameObject.AddTerrainGridSystem();
   ```

   Where myTerrainGameObject is the gameobject where your terrain is or any other gameobject is you use mesh-based terrain.

2) Or instantiate the "TerrainGridSystem" prefab from "Resources/Prefabs" and call SetTerrain(your terrain gameobject) form script.

Custom Editor Properties



Grid Configuration

This section allows you to configure the grid irrespective of its appearance or usage. Basically, these properties define the number of cells, territories and their shapes.

Terrain: assign your current terrain or root of group of objects that represent your terrain. TGS can work with Unity standard terrain or with any other mesh.

When using TGS on mesh-based objects, additional options to filter the objects will be shown:

- **Objects Name Prefix:** allows you to enter a prefix string to filter the objects by name.
- **Objects Layer Mask:** optional layer mask filter.
- **Search Global:** when enabled, objects that meet the criteria above in the entire scene will be included. Otherwise, only children of the gameobject selected will be included.

- **Topology**: this parameter defines the grid overall shape. You can choose between irregular (which uses a highly optimized voronoi tesselation algorithm), boxed or hexagonal type.

- **Territories and Cells:** each grid has a number of cells and OPTIONALLY a number of territories. A territory contains one or more cells, so you will always have more cells than territories (or same number).

  Note that adding territories will make the system slower with a high number of cells. To prevent lag during runtime while creating the neccesary meshes you can pre-warm the geometry using the API. More on that later.

- **Curvature**: if your grid has 100 or less cells, you can apply an optional curvature. Note that using this parameter will x3 the number of segments of the grid.

- **Relaxation:** this option will make irregular shapes (Voronoi) more homogeneous. Basically, it iterates over the voronoi calculation algorithm weighing and modulating the separation between cell centers. Each iteration will redo the voronoi calculation and even it's been optimized that means that mesh generation times will go up with each relaxation level.

- **Roughness:** this option will determine the level of gripping of the grid to the terrain. The greater the value, more abrupt the grid will appear with respect to the terrain. You usually will want to leave this value to the minimum possible unless your terrain is quite flat (in that case, increasing roughness will reduce vertex count).

- **Seed:** this parameter will allow you to recreate the grid with same disposition of cells. This option basically affects the random functionality.

- **Flat Cells:** when enabled, cells will ignore any terrain slope and will just be planar/horizontal cells.

- **Max Slope:** specifies the maximum slope or inclination of the terrain at the center of a cell. A value of 1 means no slope is considered hence cells will be drawn regardless of this value. Reduce to prevent cells hang over steep terrain or walls.

- **Max Height Difference:** specifies the maximum height difference between a cell vertices. If his height difference is greater than this value, that cell will be hidden. Similar to Max Slope but it can produce better results since it takes into account each individual vertex altitude while the Max Slope only considers the normal at the center of the cell.

- **Minimum/Maximum Altitude:** specifies the minimum/maximum altitude to draw cells. Useful to hide cells under water for example.

- **Mask:** assign a texture in this slot to define cells visibility (alpha channel is used to determine visibility, where a=0 means that cell is invisible). An invisible cell won't be highlighted unless the "Highlight Invisible Cells" option is enabled. Invisible cells do not participate in pathfinding either.

- **Territories Texture:** assign a color texture to define territories shape or cell ownership. One territory for each different color will be created. Ensure you use solid colors (no smooth or gradients) in the texture. If you use Gimp, you can switch to Indexed Color and set a maximum number of colors, then switch back to RGBA and export the texture in PNG format. Optionally specify which color will be used as neutral, meaning those cell won't belong to any territory.

Grid Positioning

As the grid is a phisically generated mesh, it needs to cover the terrain subtly and effectively. This means that there should be an adequate offset between the terrain geometry and that of the grid. But at the same time that offset should be the minimum possible so the grid stays below any content put over the terrain.
To achieve the best possible effect, Terrain Grid System includes several parameters that allows you to control precisely how the grid is positioned with respect to the terrain mesh.

- **Mesh Pivot**: this parameter only appears when using TGS with custom meshes. By default, TGS expects the mesh local center to be 0,0,0 but some meshes may start off another local position. In this case, adjust the mesh pivot position until the grid shows centered on the custom mesh.

- **Grid Center / Scale**: controls the size and scale of the grid on the surface its cover originally. For example, instead of covering th entire terrain, you can use these center/scale options to show the grid around a specific location of the terrain (see demo scenes for examples).

- **Depth Offset**: this parameter control how much offset is applied to the shaders used by Terrain Grid System. This value affects the Z-Buffer position of the pixels of both highlight surfaces (**Colored Depth Offset**) and grid mesh (**Mesh Depth Offset**).

- **Elevation** and **Elevation Base**: both values are summed to calculate the amount of displacement along the Y-axis. **Elevation Base** uses a numeric input entry format in the inspector instead of a slider to allow finer/greater height values.

- **Min Elevation Multiplier**: mesh root gameobject, cells layer and territories layer are separated by a very small offset (about 0.01f) in the Y-axis. You can control the size of this offset with this multiplier. Usually, this value should be left at 1 but if you use a terrain mesh with a very big scale, this offset can result in a big shift upwards so this parameter can be useful to reduce that gap.

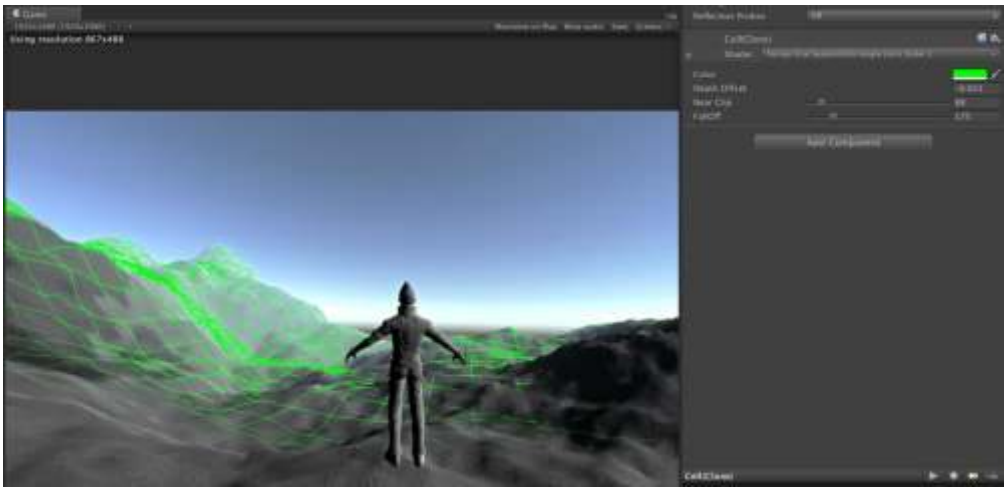
- **Camera Offset**: apply a displacemenet of the grid towards the camera position. This displacement is dynamic, meaning that it will move the grid automatically whenever the grid or the camera changes position.

- **Normal Offset**: this option will apply a displacement per vertex along the normal of the terrain below its position. It will make the grid "grow" around the terrain in all directions. Note that this is an expensive operation. You usually don't need to use this parameter.

Grid Appearance

This section controls the look and feel of the grid system and it should be self-explanatory. However it's important to know that activating either "Show Territories" or "Colorize Territories" (or set the highlight mode to Territory as well) will enable the territory mesh generation. So, if you don't use the territory functionality, make sure all options regarding territories are unchecked to save time and memory.

Grid lines are drawn using custom shaders that allow you to specify a near clip and falloff. These properties should be useful for those situations where the camera is really near the grid (like in FPS):



To change the default clipping and falloff values, use the Near Clip Fade and Near Clip Fade FallOff sliders.

The Cell Fill Padding option can be used to enlarge or reduce the surface of a cell when it's colored or textured.

Grid Behaviour

This section controls what the grid can do.

The first and most important field is the Terrain reference, which you should set to make the grid system start working with your terrain.

In addition to that, you may choose:

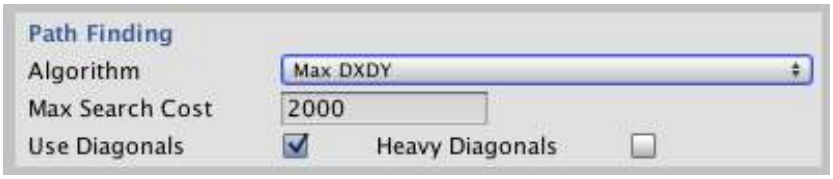
- **Selection Mode**: choose between None, Cells or Territories.

- **Overlay Mode**: choose between Ground or Overlay modes. The overlay mode will make the highlight surfaces to be displayed on top of everything. Not very fancy but it's the fastest and prevents any visual artifact with complex terrains. The default option, Ground, will make the highlight surfaces to appear below any content put over the terrain, effectively seeming that it's being painted over the terrain.

- **Respect Other UI**: when enabled, grid interaction is ignored while pointer is over a UI element (only uGUI elements are supported, those under a Canvas root element, not immediate GUI or elements created from OnGUI() functions).

Path Finding

This section controls the path finding behaviour.



- **Algorithm**: choose among available algorithms:
  - **MaxDXDY**: use estimations based on horizontal/vertical distance to target. Produces more natural paths althouth not optimal.
  - **Manhattan**: produces jaggy paths.
  - **Diagonal Shortcut**: favours diagonals in path.
  - **Euclidean**: estimation based on the straight distance to target. Produces the optimal path.
  - **Euclidean Non SQR**: similar but don't uses sqrt which makes it faster.
  - **Custom1**: experimental method.

- **Max Search Cost**: the maximum path length.
- **Use Diagonals**: if diagonals between cells can be used.
- **Heavy DIagonals**: add an extra cost for diagonals, making them less frequent.

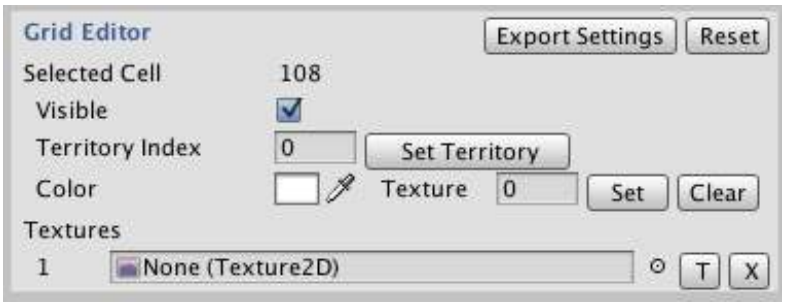
The Path Finding algorithm is very fast but TGS also includes an optimized variant of the algorithm for grids that have column count power of 2 (4, 8, 16, 32, 64, 256, …).
Check demo scene 12 for example on using Path Finding with Terrain Grid System. Also read the available APIs for using path finding feature below.

TGS provides you 4 options to define available paths:

- **CellSetCanCross**: calling this method you can simply specify if a cell blocks any path or not. The block state can be used as a parameter in methods like FindPath or CellGetNeighbours, etc.
- **CellSetGroup**: use this method to assign a cell to a custom group. You can later specify which groups are allowed when computing a path calling FindPath method.
- **CellSetCrossCost**: defines a custom crossing cost when entering a cell.
- **CellSetSideCrossCost**: defines a custom crossing cost for a givel cell and a specific edge with option to consider moving direction (ie. entering or exiting the cell through that edge).

Grid Editor

This section allows you to customize the cells in Editor time.



With the Scene View selected (not Game View), you can use the mouse to select any cell and change their properties.

Export Settings will create add a “TGS Config” script which serves as a container for all the settings and will load them when activated.

Reset will clear all cell settings and reverts them to default values.

Textures array is a convenient way to load several textures and use them quickly to paint several cells. Just click the “T” small buton to enable the “paint mode”. Once enabled, you can click several cells to assign the chosen texture.

You can have several “TGS Config” scripts added. To load a specific configuration just enable the desired TGS Config component or call “LoadConfiguration()” method of that script.

Positioning the Grid on your terrain

Placing the grid at the center of the terrain and making it fit the entire terrain may be not a good idea if your terrain is very big. In this case and depending on your requirements, it's possible that you will need to draw a lot of cells. Although this can be done just entering the proper number of rows and columns in the inspector, it's not performance wise.

A solution for big terrains is just draw a portion of the grid around the character. For that, you need to set **gridCenter** and **gridScale** properties.

- GridCenter values range from -0.5 (left most side of the terrain) to 0.5 (right most side of the terrain). By default gridCenter is set to 0,0, which matches the center of the terrain. Setting it to -0.5, 0.5 for example will center the grid on the top/left corner of the terrain.
- GridScale values range from 0 to 1 and are relative to the terrain size. So a scale of 0.1,0.1 will make the grid a 10% of the size of the terrain.

Another useful property is gridCenterWorldSpace which lets you assign a world space position directly and the grid will be displayed around that point. For example you can assign the character transform.position to this property.

Check out demo scenes 10 and 10b for useful examples.

Changing default Input system

Terrain Grid System was designed to use the classic Unity input system (like Input.GetMouseButton methods, etc.). To support future and new input systems, the asset has decoupled the input calls and centralized them into a different class, called DefaultInputSystem.cs located in TerrainGridSystem/Scripts/Core/Input folder. This class implements the interface IInputProxy.

There's another class called NewInputSystem.cs which adds support for the New Input System. Terrain Grid System will automatically use one or the other input proxy class depending on the current input system active in Player Settings.

You can also modify these classes or create a new class that derive and implements the IInputProxy, so you provide your own input logic for common tasks like getting the key or button pressed.

Once you have created your own class, you can just assign it to Terrain Grid System component at runtime like this:

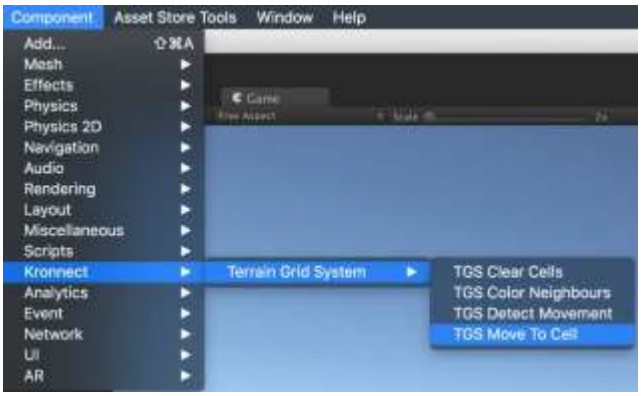
```
TerrainGridSystem tgs = TerrainGridSystem.instance;
MyCustomInputSystem newInput = new MyCustomInputSystem();
tgs.input = newInput;
```

Programming Guide (API)

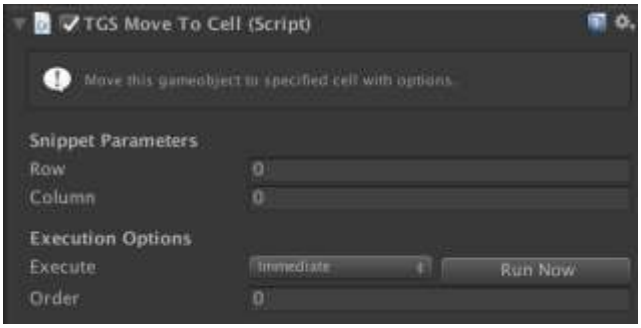
Note for new developers!
If you're new to programming, you may find overwhelmed by the large list of C# functions and properties. We have created a few useful scripts for new developers called "Snippets". They're scripts you can add to gameobjects to perform simple tasks, like "move this object to target cell" without coding.

These snippets are just fancy wrappers for the API provided by Terrain Grid System so you can take a look into them and learn how do they use the API.

The available snippets can be found in the Component menu:



For example, let's say you have a gameobject you want to move on the grid. You can add the script "TGS Move To Cell" to your gameobject and enter the row and column in the inspector:



If you would like more snippets or have any question please send us an email or use our support forum. We'll gladly add more snippets for you ☺

Important! These snippets just provide like the 1% of functionality provided by the API and are meant to introduce you into some common features. **If you already know how to program in C# then you can go directly to the next section and learn the API** (it's not that complex after all!).

Use Terrain Grid System from scripting

You can instantiate the prefab "TerrainGridSystem" or add it to the scene from the Editor as explained in the "How to use this asset in your project" section. Once the prefab is in the scene, you can access the functionality from code through the static instance property:

```
using TGS;

TerrainGridSystem tgs;

void Start () {
   tgs = TerrainGridSystem.instance;
            ...
}
```

To get the cell beneath a gameobject in your scene use CellGetAtPosition and pass the world space coordinate:

```
Vector3 characterPosition = transform.position;
Cell cell = tgs.CellGetAtPosition(characterPosition, true);
int row = cell.row;
int column = cell.column;
```

To fade surrounding cells:

```
List<Cell>neighbours = tgs.CellGetNeighbours(sphereCell);
foreach(Cell cell in neighbours) {
    tgs.CellFadeOut(cell, Color.red, 2.0f);
}
```

To fade cells by row and column:

```
const int size = 3;
for (int j=sphereCell.row - size; j<=sphereCell.row + size;j++) {
      for (int k=sphereCell.column - size; k<=sphereCell.column + size; k++) {
          int cellIndex = tgs.CellGetIndex(j,k, true);
          tgs.CellFadeOut(cellIndex, Color.blue, 1.0f);
      }
}
```

Take a look at the demo scenes included in the asset for more sample code!

Public API structure

All TerrainGridSystem API is exposed through TerrainGridSystem class. This is a single component that has been split for convenience in several files located in TerrainGridSystem/Scripts folder:

- TerrainGridSystem: contains generic functions and properties.
- TGSCells: all cell-related.
- TGSTerritories: all territory-related.
- TGSPathfinding: pathfinding stuff.
- TGSConfig: this is the component that stores the cell configuration when you use and export the cell setup using the Grid Editor.

Note that the above are just filenames, but the class is the same “TerrainGridSystem”. This approach follows the partial class standard in C# to make easy the reorganization of “God” (or big) utility classes. This means that you will be just using the traditional gameObject.GetComponent<TerrainGridSystem>() or just TerrainGridSystem.instance to get a reference to the API, irrespective of the number of classes files.

Thanks to this file organization you can easily check the different public properties and functions of the API by category (cell-related, territory-related, …)

Complete list of public properties, methods and events

Basic grid functions

tgs.SetTerrain(gameObject): specifies the terrain gameobject linked to the grid. Usually this is set in Editor time but you can also call thi method if you instantiate the prefab at runtime. Instead of instantiating the prefab you can just call AddTerrainGridSystem() directly on any gameobject including your terrain gameobject (AddTerrainGridSystem is an extension method for GameObject class).

tgs.numCells: desired number of cells for irregular topology. The actual generated cells will vary depending on the topology.

tgs.ColumnCount: number of columns for boxed and hexagonal types.

tgs.RowCount: number of rows for boxed and hexagonal types.

tgs.SetDimensions(int rows, int columns, bool keepCellSize): sets the number of rows and columns in one step (only hexagonal or box grid topology). This is faster than settings rowCount and columnCount separately. The parameter keepCellSize determines if the individual cell size is preserved (only valid for hexagonal and box topologies).

tgs.seed: seed constant for the random generator.

tgs.gridCenter: the center of the grid with respect to its parent (0,0 = center).

tgs.gridCenterWorldPosition: sets or retrieves the center of the grid in world space coordinates. You can also call **tgs.SetGridCenterWorldPosition** method instead.

tgs.gridScale: the scale of the grid with respect to its parent (1,1 = full terrain/grid size)

tgs.cellSize: directly sets the individual cell size (Vector2). Setting this property will recalculate the grid scale accordingly.

tgs.gridMask: optional texture used to defined cells visibility. The texture is matched against the whole grid parent surface. An alpha value of 0 means cell will be invisible.

tgs.gridTopology: type of cells (Irregular, Box, Hexagonal, …)

tgs.gridRelaxation: relaxation factor for the irregular topology (defaults 1). It creates more homogeneous irregular grids but has a performance hit during the creation of the grid (not afterwards).

tgs.gridCurvature: will bend cell edges for all topologies. It will add more vertex count to the mesh so it's only available on grids with less than 100 cells.

tgs.terrainCenter: returns the world space position of the center of the terrain.

tgs.bounds: the bounds of the grid in world space. This property does not account for height.

tgs.gridElevation: vertical offset of the grid with respect to the terrain for fine-tuning z-fighting issues.

tgs.gridElevationBase: vertical position for the grid, used to elevate grid over terrain. Both gridElevation and gridElevationBase are added when computing the position of the grid. Use gridElevationBase if you want the grid to float above the terrain and gridElevation for small height adjustments.

tgs.gridCameraOffset: dynamic camera-oriented displacement. This will move the mesh towards the camera.

tgs.gridNormalOffset: general offset of the mesh from the terrain. This will make the grid "grow" in all directions. This will impact performance during the mesh creation.

tgs.gridDepthOffset: Z-Buffer offset for the grid system shaders. Very cheap method to prevent z-fighting which can be combined with above methods. You should use this property first to adjust your grid.

tgs.gridRoughness: gripping factor for the mesh. The lower the value the better fit of the mesh to the terrain.

tgs.nearClipFade: distance from the camera for the fade in effect of the grid. Useful in first-person-view games.

tgs.nearClipFadeFallOff: falloff or gradient amount for the fade in effect.

tgs.voronoiSites: optionally set a list of starting Voronoi site positions. The list will be completed up to numCells if voronoiSites list provided has less items.

Cell related

tgs.cells: return a List<Cell> of all cells/states records.

tgs.showCells: whether to show or not the cells borders.

tgs.cellHighlighted: returns the cell/state object in the cells list under the mouse cursor (or null if none).

tgs.cellHighlightedIndex: returns the cell/state index in the cells list under the mouse cursor (or -1 if none).

tgs.cellHighlightNonVisible: sets if invisible cells should be highlighted when pointer is over them (default is true).

tgs.cellLastClickedIndex: returns the index of last cell clicked.

tgs.cellHighlightColor: color for the highlight of cells.

tgs.cellHighlightNonVisible: set to true if you want to be able to highlight invisible cells.

tgs.cellBorderColor color for all cells borders.

tgs.cellBorderAlpha: helper method to change only the alpha of cell borders.

tgs.CellSetTag(int cellIndex, int tag): assigns a cell a user-defined integer tag. Cell can be later retrieved very quickly with CellGetWithTag.

tgs.CellGetWithTag(int tag): retrieves a cell previously tagged with an integer using the method CellSetTag.

tgs.CellGetIndex(Cell cell): returns the index of the provided cell object in the Cells collection.

tgs.CellGetIndex(row, column, clampToBorders): returns the index of the provided Cell by row and column. Use clampToBorders = true to limit row/column to edges of the grid or to allow wrap around edges otherwise.

tgs.CellGetPosition(int cellIndex): returns the center of a cell in the world space.

tgs.CellGetRect(int cellIndex): returns the 2D rectangle enclosing the cell in local space (-0.5..0.5 range).

tgs.CellGetRectWorldSpace(int cellIndex): returns the 3D bounds of the cell in world space.

tgs.CellGetVertexCount(int cellIndex): returns the number of vértices of any cell.

tgs.CellGetVertexPosition(int cellIndex, int vertexIndex): returns the world space position of any vertex of a cell.

tgs.CellGetAtPosition(Vector3 position, Vector3 wordSpace): returns the cell that contains position (in local space coordinates which ranges from -0.5 to 0.5 or from a point in world space).

tgs.CellGetAtPosition(int column, int row): returns the cell located at given row and col (only boxed and hexagonal topologies).

tgs.CellGetInArea(Bounds bounds, List<int>indices, float padding = 0, bool checkAllVertices = false)
Returns a list of cell indices contained in the bounds or under those bounds (eg. under a gameobject's collider bounds). Padding is an optional margin that adds or substract to the boundary. When checkAllVertices is true, not only the center but all cell vertices are checked.

tgs.CellGetInArea(Collider collider, List<int>indices, int resolution, float padding = 0, float offset = 0)
Returns a list of cell indices contained under a collider. Resolution and padding are used to defined the granularity of the algorithm which tests cells inside the projected collider bounds. Offset can be used to modify the projection axis.

tgs.CellGetWithinCone(int cellIndex, Vector2 direction, float maxDistance, float angle, List<int> cellIndices)
Returns a list of cells contained in a cone defined by a starting cell, a direction, max distance and angle in degrees (max distance is defined in local space coordinates of grid being the borders at -0.5 and 0.5)

tgs.GetCellsWithinCone(int cellIndex, int targetCellIndex, float angle, List<int> cellIndices)
Returns a list of cells contained in a cone defined by a starting cell, target cell and angle in degrees.

tgs.CellGetNeighbour(int cellIndex, int side): returns the index of the adjacent cell by its side name (top, right, bottom left, …).

tgs.CellGetNeighbours(int cellIndex, …): returns a list of neighbour cells to a specific cell optionally specifying the maximum number of steps, maximum total cross cost, cell group mask and other options.

tgs.CellGetNeighboursWithinRange(int cellIndex, …): similar to CellGetNighbours method but accepts min and max distance.

tgs.CellGetHexagonDistance(index1, index2): returns the number of steps to reach cell2 from cell1 in an hexagonal grid. This method does not take into account cell groups nor blocking cells.

tgs.CellMerge(Cell 1, Cell 2): merges two cells into the first one.

tgs.CellSetTerritory(int cellIndex, int territoryIndex): changes the territory of a cell and updates boundaries. Call Redraw() to update the grid afterwards.

tgs.CellToggleRegionSurface(cellIndex, visible, color, refreshGeometry): colorize one cell's surface with specified color. Use refreshGeometry to ignore cached surfaces (usually you won't do this).

tgs.CellToggleRegionSurface(cellIndex, visible, color, refreshGeometry, texture): color and texture one cell's surface with specified color. Use refreshGeometry to ignore cached surfaces (usually you won't do this). Texture can be scaled, panned and/or rotated adding optional parameters to this function call.

tgs.CellToggleRegionSurface(int cellIndex, bool visible, Color color, bool refreshGeometry, Texture2D texture, Vector2 textureScale, float textureRotation, bool overlay): same but allows to specify a texture scale and rotation and optionally if the colored surface will be shown on top of objects.

tgs.CellHideRegionSurface(int cellIndex): uncolorize/clears one cell's surface.

tgs.CellHideRegionSurfaces(): uncolorize/clears all cells.

tgs.CellFadeOut(int cellIndex, color, duration, repetitions): colorizes a cell and fades it out for duration in seconds.

tgs.CellFlash(int cellIndex, color, duration, repetitions): flashes a cell from given color to default cell color for duration in seconds.

tgs.CellBlink(int cellIndex, color, duration, repetitions): blinks a cell from current color to a given color and back to original cell color for duration in seconds.

tgs.CellColorTemp(int cellIndex, color, duration): temporarily colors a cell or group of cells.

tgs.CellCancelAnimations(cellIndex): removes any effect on a cell or group of cells.

tgs.CellGetColor(cellIndex): returns current cell's fill color.

tgs.CellGetTexture(cellIndex): returns current cell's fill texture.

tgs.CellSetTexture(cellIndex, texture): sets the texture for a cell. Similar to CellToggleRegionSurface.

tgs.CellSetColor(cellIndex, color): sets the color for a cell. Similar to CellToggleRegionSurface.

tgs.CellSetMaterial(cellIndex, material): assigns user defined material to a cell.

tgs.CellGetNormal(cellIndex): returns the terrain normal at the center of the cell.

tgs.CellIsBorder: returns true if the cell sits at the edges of the grid.

tgs.CellIsVisible: returns true if the cell is visible.

tgs.CellSetVisible: makes a cell or group of cells visible or invisible.

tgs.CellSetBorderVisible: makes a cell's border visible or invisible.

tgs.CellGetGameObject: returns the cell surface gameobject.

tgs.CellScaleSurface: applies a scale to a colored/textured cell surface.

tgs.CellRemove: completely removes a cell from the grid. Call Redraw() to update the grid afterwards.

tgs.CellSetCanCross: specifies if a cell can be part fo a route returned by Pathfinding methods. This method is used to specify blocking cells.

tgs.CellSetGroup: assigns a cell to a group. Groups can be used to use certain cells or not when calling FindPath, CellGetNeigbhours or CellGetLineOfSight. Using groups you can create different kind of blocking cells. See demo scene 12 for an example.

tgs.CellGetGroup: gets the group of a cell.

tgs.CellGetFromGroup: gets all cell indices from a group.

tgs.CellSetSideCrossCost/CellGetSideCrossCost: sets or obtain the crossing cost for a cell and a specific edge with option to apply that cost to any direction (ie. existing or entering the cell through that edge).

tgs.CellSetAllSidesCrossCost: convenient method to quickly assign a crossing cost to a cell regardless of the edge used to travel.

tgs.CellGetConfigurationData: retrieves a string-packed description of the cells properties. Use CellSetConfigurationData to load the configuration.

tgs.CellSetConfigurationData: loads a configuration of cells from a string.

tgs.CellGetSettings: similar to CellGetConfigurationData but returns an array of TGSConfigEntry values (one per cell). A TGSConfigEntry is a struct that contains important attributes about each cell.

tgs.CellSetSettings: similar to CellSetConfigurationData but accepts an array of TGSConfigEntry values.

tgs.CellAddSprite(int cellIndex, Sprite sprite, bool adjustScale): creates a gameobject with a given sprite and positions/scale it to match a given cell.

Territory related

tgs.numTerritoriess: desired number of territories (1-number of cells).

tgs.territories: return a List<Territory> of all territories.

tgs.showTerritories: whether to show or not the territories borders.

tgs.showTerritoriesOuterBorder: whether to show or not the territories perimeter around the grid.

tgs.territoryHighlighted: returns the Territory object for the territory under the mouse cursor (or null).

tgs.territoryHighlightedIndex: returns the index of territory under the mouse cursor (or -1 if none).

tgs.territoryLastClickedIndex: returns the index of last territory clicked.

tgs.territoryHighlightColor: color for the highlight of territories.

tgs.territoryFrontiersColor: color for all territory frontiers.

tgs.territoryDisputedFrontiersColor: color for shared frontiers between two territories.

tgs.territoryFrontiersAlpha: helper method to change only the alpha of territory frontiers.

tgs.colorizeTerritories: whether to fill or not the territories.

tgs.colorizedTerritoriesAlpha: transparency level for colorized territories.

tgs.TerritoryToggleRegionSurface(territoryIndex, visible, color, refreshGeometry): colorize one territory's surface with specified color. Use refreshGeometry to ignore cached surfaces (usually you won't do this).

tgs.TerritoryToggleRegionSurface(territoryIndex, visible, color, refreshGeometry, texture): color and texture one territory's surface with specified color. Use refreshGeometry to ignore cached surfaces (usually you won't do this). Texture can be scaled, panned and/or rotated adding optional parameters to this function call.

tgs.TerritoryToggleRegionSurface(cellIndex, visible, color, refreshGeometry, texture, textureScale, textureRotation, overlay): same but allows to specify a texture scale and rotation and optionally if the colored surface will be shown on top of objects.

tgs.TerritorySetMaterial(territoryIndex, material): assigns user defined material to a territory.

tgs.TerritoryHideRegionSurface(territoryIndex): uncolorize/clears one territory's surface.

tgs.TerritoryGetNeighbours(territoryIndex): returns a list of neighbour territories to a specific territory.

tgs.TerritoryGetCells(territoryIndex, regionIndex): returns a list of cells belonging to a territory or any of its regions.

tgs.TerritoryGetAtPosition(position): returns the territory that contains position (in local space coordinates).

tgs.TerritoryGetPosition(territoryIndex): returns the center of the territory in world space.

tgs.TerritoryGetRectWorldSpace(territoryIndex): returns the boundaries of the territory in world space coordinates.

tgs.TerritoryGetVertexCount(territoryIndex): returns the number of points of the territory polygon.

tgs.TerritoryGetVertexPosition(territoryIndex, vertexIndex): returns the position of the specified vertex in world space coordinates.

tgs.TerritoryFadeOut(territoryIndex, color, duration, repetitions): colorizes a territory and fades it out for duration in seconds.

tgs.TerritoryFlash(territoryIndex, color, duration, repetitions): flashes a territory from given color to default territory color for duration in seconds.

tgs.TerritoryBlink(territoryIndex, color, duration, repetitions): blinks a territory from current color to a given color and back to original territory color for duration in seconds.

tgs.TerritoryColorTemp(cellIndex, color, duration): temporarily colors a territory or group of territories.

tgs.TerritoryCancelAnimations(territoryIndex): removes any effect on a territory.

tgs.TerritoryIsVisible(territoryIndex): returns true if territory is visible.

tgs.TerritorySetVisible(territoryIndex): makes a territory visible or invisible.

tgs.TerritorySetBorderVisible: makes a territory's border visible or invisible.

tgs.TerritorySetFrontierColor(territoryIndex, color): assigns a new color for the frontiers of a given territory.

tgs.CreateTerritories(texture, neutral): generate territories based on distinct colors contained in the provided texture ignoring neutral color.

tgs.TerritorySetNeutral(territoryIndex, neutral): specifies if a given territory should be considered as neutral. Neutral territories do not dispute frontiers.

tgs.TerritoryIsNeutral(territoryIndex): returns true if the given territory is neutral.

tgs.TerritoryGetFrontier(territoryIndex, regionIndex): returns a list of points (Vector2) that form a territory frontier.

tgs.TerritoryGetFrontierCells(territoryIndex, otherTerritoryIndex, ref List<int> cellIndices): returns a list of cells that form a territory frontier. Optionally restrict to frontier with another territory index.

tgs.TerritoryGetGameObject: returns the territory's surface gameobject.

tgs.TerritoryScaleSurface: applies a scale to a colored/textured territory surface.

tgs.TerritoryDrawFrontier(territoryIndex, adjacentTerritoryIndex, material, color): draws the frontier around the territory specified by territoryIndex. Optional parameters include: adjacentTerritoryIndex (only draw frontier shared by those two territories), a material or a color.
Note: this method returns the gameobject for the newly drawn frontier. You're responsible of disposing this gameobject or redrawing it since TGS won't refresh it nor destroy it when changing the grid.

User interaction / Highlighting

tgs.highlightMode: choose between None, Territories or Cells. Defaults to Cells.

tgs.highlightFadeAmount: alpha range for the fading effect of highlighted cells/territories (0-1).

tgs.highlightMinimumTerrainDistance: minimum distance to the terrain to allow highlighting of cells/territories.

tgs.HighlightTerritoryRegion(territoryIndex, refreshGeometry, regionIndex): manually highlights a territory region. RefreshGeometry can be used to force creation of the highlight mesh (usually not required). RegionIndex can be used to highlight a specific region of a territory if it's split in several zones (usually not required).

tgs.HideTerritoryRegionHighlight(territoryIndex): hides the highlight of the territory.

tgs.HighlightCellRegion(cellIndex, refreshGeometry): manually highlights a cell region.

tgs.HideCellRegionHiglight(cellIndex): hides the highlight of the cell.

tgs.overlayMode: decide whether the highlight is shown always on top or takes into account z-buffer which will make it more fancy and part of the scene.

tgs.localCursorPosition: the current cursor position on the grid (mouse over the grid) in local grid coordinates. Use tgs.transform.TransformPoint(x) method to convert to world space coordinates.

tgs.localCursorLastClickedPosition: the position of the last click on the grid in local grid coordinates. Use tgs.transform.TransformPoint(x) method to convert to world space coordinates.

Path Finding & LOS related

tgs.CellGetSideBlocksLOS(cellIndex, side, blocks): returns whether a cell's side blocks line of sight.

tgs.CellSetSideBlocksLOS(cellIndex, side, blocks): specify whether a cell's side blocks line of sight.

tgs.CellSetCanCross: specifies if this cell blocks any path.

tgs.CellSetCrossCost: specifies the cost to enter a cell (defaults to 1).

tgs.CellSetSideCrossCost: specifies the cost to enter or exit a cell through a specific edge.

tgs.FindPath(cellIndexStart, cellIndexEnd): returns a list of cell indices that form the path.

tgs.FindPath(cellIndexStart, cellIndexEnd, out totalCost, maxSearchCost, maxSteps): returns a list of cell indices that form the path and the totalCost for the path. Optionally pass a maxCost and maxSteps (use OnCellCross event to return custom costs for any cell).

tgs.FindPath(cellIndexStart, int cellIndexEnd, List<int> cellIndices, out float totalCost, float maxSearchCost = 0, int maxSteps = 0, int cellGroupMask = -1, CanCrossCheckType canCrossCheckType = CanCrossCheckType.Default, bool ignoreCellCosts = false, bool includeInvisibleCells = true): returns a list of cell indices that form the optimal path between two cells with extra parameters:

- **cellGroupMask**: bitwise mask to filter which cells can be included in the path based on their group value (set any cell group value using CellSetGroup method).
- **canCrossCheckType**: specifies how "canCross" flag of each cell should be used. This parameter let you include or exclude start or end cells that has canCross disabled. A cell can be set to "block" by using CellSetCanCross method.
- **ignoreCellCosts**: ignores any custom cost assigned to a cell.
- **includeInvisibleCells**: if hidden cell can be used in the path.

tgs.CellGetLineOfSight(startPosition, endPosition, …): returns true or false if there's a line of sight from the starting cell to target cell. It also returns the list of cells in the line as well as the world positions.

tgs.CellGetLine(startPosition, endPosition, …): returns a list of cells that form a line between two cells with options.

tgs.CellTestLineOfSight(startIndex, indices, …): removes any cell index from the indices list that are not visible from the given starting cell.

Other useful instance methods

tgs.instance: reference to the actual Terrain Grid System component on the scene.

tgs.HideAll: hides all surfaces and clears the internal cache (cached surfaces are kept in memory).

tgs.ClearAll: hides all surfaces and clears the internal cache (surfaces are destroyed and memory freed, and will need to be generated again if a cell/territory is colored/textured).

tgs.Redraw: forces a redraw of the grid mesh. Optionally set the parameter reuseTerrainData to true to speed up computation if terrain has not changed.

tgs.RedrawCells(List<Cell>): forces a redraw of specific cells. This is faster than redrawing the entire grid when only a subset of cells is required.

tgs.CancelAnimations: similar to CellCancelAnimations or TerritoryCancelAnimations but stops any animation over the grid (flash, blink, fadeout, color temp, etc.).

Other useful static methods

TerrainGridSystem.grids: list that contains all grids in the scene.

TerrainGridSystem.GetGridAt(position): returns the grid that contains a given position.

Events

Note: in order to support multi-grid setups, all events receive the reference to the grid as the first parameter so you always know which grid is triggering the event.

tgs.OnCellEnter: triggered when pointer enters a cell.
tgs.OnCellExit: triggered when pointer exits a cell.
tgs.OnCellMouseDown: triggered when user presses mouse button on a cell.
tgs.OnCellMouseUp: triggered when user releases mouse button on a cell.
tgs.OnCellClick: triggered when user presses and releases the mouse pointer on the same cell.
tgs.OnCellHighlight: triggered when a cell is goint to be highlighted (usually when pointer is over it). A ref parameter allows you to cancel highlight and implement your own coloring system.
tgs.OnCellDragStart: triggered when user starts dragging on a cell.
tgs.OnCellDrag: triggered while user drags.
tgs.OnCellDragEnd: triggered while user released the left button while dragging.

tgs.OnTerritoryEnter: triggered when pointer enters a territory.
tgs.OnTerritoryExit: triggered when pointer exits a territory.
tgs.OnTerritoryMouseDown: triggered when user presses mouse button on a territory.
tgs.OnTerritoryMouseUp: triggered when user releases mouse button on a territory.
tgs.OnTerritoryClick: triggered when user presses and releases mouse button on the same territory.
tgs.OnTerritoryHighlight: triggered when a territory is goint to be highlighted (usually when pointer is over it). A ref parameter allows you to cancel highlight and implement your own coloring system.

tgs.OnPathFindingCrossCell: triggered when path finding algorithm estimates cost to this cell. You can use this event to return extra cost and customize path results.
tgs.OnRectangleSelection: triggered when user completes a rectangle selection. Read “Rectangle Selections” section for details.
tgs.OnRectangleDrag: triggered when user drags a rectangle selection. Read “Rectangle Selections” section for details.

Handling cells and territories

Please, refer to demo scripts included in demo scenes for example of API usage, like selecting, merging and toggling cells visibility, …

Rectangle Selection

Demo scene 20 showcases how to use the OnRectangleSelection event.

Firstly, to enable rectangle selection you must call tgs.enableRectangleSelection = true. At this point, user will be able to click and drag a rectangle. When the user drags a selection, the even OnRectangleDrag fites. And when he/she completes the rectangle selection (by releasing the mouse button), the event OnRectangleSelection is fired:

The following code corresponds to demo scene 20. It allows the user to select a rectangle of cells and color them in red:

```
using System.Collections.Generic;
using UnityEngine;
using TGS;

public class Demo20 : MonoBehaviour {

    TerrainGridSystem tgs;

    void Start() {
        tgs = TerrainGridSystem.instance;
        tgs.enableRectangleSelection = true;
        tgs.OnRectangleSelection += SelectCells;
        tgs.OnRectangleDrag += HighlightRectangleArea;

        tgs.OnCellClick += (grid, cellIndex, buttonIndex) => PaintCell(cellIndex);
    }

    void PaintCell(int cellIndex) {
        tgs.CellSetColor(cellIndex, Color.red);
    }

    void SelectCells(TerrainGridSystem grid, Vector2 localPosStart, Vector2 localPosEnd) {
        List<int> cells = new List<int>();
        tgs.CellGetInArea(localPosStart, localPosEnd, cells);

        // Hides all surfaces (and clear cache)
        tgs.ClearAll();

        // Color selection
        tgs.CellSetColor(cells, Color.yellow);
    }

    void HighlightRectangleArea(TerrainGridSystem grid, Vector2 localPosStart, Vector2 localPosEnd) {
        List<int> cells = new List<int>();
        tgs.CellGetInArea(localPosStart, localPosEnd, cells);

        // Hides all surfaces (but keep them in cache)
        tgs.HideAll();

        // Color current selection
        tgs.CellSetColor(cells, new Color(0, 0, 1, 0.25f));
    }

}
```

Custom user data

There are 3 ways you can add custom data to cells and territories:

Option 1

Using the attrib property. Every cell or territory object has an “attrib” property which is a JSON object that can store any number of custom fields.

For example, you coud use:
tgs.cells[10].attrib[“resource”] = “coal”

This will store the word “coal” associated to the key “resource” of cell with index 10.

Option 2

Using the tag property. Each cell has a “tag” property which is a basic integer. You can use this field for any purpose like storing the id of some other object that’s managed by you.

Option 3

Extending the cell or territory class. Both classes are defined with the C# partial keyword which allows you to add custom fields or methods in another file. The compiler will simply merge the contents of both files when compiling the class. The benefit of adding your custom fields/properties in a different file is that it doesn’t get overwritten when you upgrade to a newer version of the asset.