WORLD MAP STRATEGY KIT MANUAL

Table of Contents

Introduction

Thank you for purchasing!

World Map Strategy Kit (WMSK) is a commercial asset for Unity that allows to dramatically speed up the development of strategy/RTS/world map games. WMSK is packed with exciting features:

- Ready to use dataset with frontier data of 241 countries, +4000 provinces and +7100 most important cities in the world.
- Ability to colorize and also highlight the regions of countries and provinces/states as mouse hovers them. Per country texture support!
- Automatically draws country labels, with placement options.
- Define custom mount points and customize its location and tags with the editor. Mass mount point tool to randomly populate countries and continents with your resources or special locations.
- Viewport rendering targets (supports cropping) including 3D surface with heighmap, fog of war and cloud layer.
- Terrain mode: render WMSK's visual features (country labels, frontiers, regions) onto Unity terrain.
- Quickly locate and center any country, city, province or custom location.
- Imaginary lines: draw custom latitude, longitude and cursor lines.
- Ease choose beteween different catalogs included based on quality/size for frontiers. Filter number of cities by population and/or type (normal cities and region/country capitals)
- Lots of customization options: colors, labels, frontiers, provinces, cities, Earth (several styles including 16K high resolution mode)…
- Hexagonal grid and path-Finding functionality included for getting routes from any two points of the map. Units can be assigned terrain capabilities and the system will determine an optimal route for them automatically when moving to a destination.
- Comprehensive API and extra components: **Calculator**, **Tickers**, **Decorator** and **Map Editor**.
- Dedicated and responsive support forum.

Support & Contact info

We hope you find the asset easy and fun to use. Feel free to contact us for any enquiry and please remember to **rate this asset on the Asset Store** – that encourages us to continue investing efforts in future updates!

Visit our Support Forum for usage tips and access to the latest beta releases.

Kronnect
Email: contact@kronnect.me
Kronnect Support Forum: http://www.kronnect.me

System Requirements

WMSK requires Unity 2018.4 (LTS) or later.
It works with standard/builtin and Universal Rendering Pipeline.

Additional setup for Universal Rendering Pipeline (URP)

1) Demo scenes are designed for the standard/builtin pipeline. When using URP, select top menu Edit / Rendering Pipeline / Universal Rendering Pipeline / Upgrade Project Materials option. This will upgrade all materials using the standard shader to the URP Lit shader.
2) If you want to use the terrain mode, please import the package located in WMSK / Resources / WMSK / Shaders / LWRP / TerrainShaders folder. This package contains native URP compatible shaders for terrain, so WMSK features can be visible in this mode.

Where to start?

WMSK is aimed to help you create games based on world map, from simple country quiz games to complex AAA strategy or tactical games.

We recommend using the following course to learn and use WMSK:

1. **Run the demo scenes**
2. **Read this documentation**
3. **Make the game design key choices**
4. **Check out the demo scenes that match your game design and learn by example**

Running the demo scenes

The best way to start learning about WMSK features is to play with each of the demo scene included in the asset in sequential order:

1. First, create an empty project and import the asset.
2. You'll find several demo scenes inside the "Demos" folder. Start with demo scene #1, run it and experiment with the different options presented. Each demo scene contains a C# file with the code behind that demo interface – we recommend you take a look at that code to learn how the API is used.
3. Once you finish demo scene #1, follow with the demo scene #2 and so on, until you finish all demo scenes.

Each demo scene contains a WorldMapStrategyKit 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 and methods of the asset from code (C#).

Now that you have explored the demo scenes included, continue with the next section which will propose different game design options for using WMSK.

Game design key choices

Once you have played with the different demos, before starting your own game with WMSK you need to think about two main topics:

- Decide the world structure (eg. will my game include provinces or just countries? Will I make my own map or use the provided world map?).
- Choose one of the 3 available representations: normal 2D flat map, viewport or terrain mode.

World Structure

WMSK provides complete country and provinces frontiers, as well as a large catalogue of most important cities around the world. From a strategy game design perspective, there are several options to leverage the content provided in the asset:

Option 1: Countries only

In this mode, provinces and cities are hidden and players are offered a world map based on countries where they can select and interact with those selections (quiz games, global strategy games, …).

You can use the events OnCountryEnter, OnCountryClick, OnCountryExit for example, to react to player interaction. To illustrate the ownership of a country you could either colorize then, using methods like ToggleCountrySurface, or adding markers (like flag sprites) on the center of each country.

In this mode, both high and low resolution frontiers can be used. For older mobile devices you may want to use the low-resolution frontiers.

Option 2: Provinces only

In this mode, provinces are the key to the game. Countries are visible but they don't highlight when player moves the pointer. As with countries, you can make use of events like OnProvinceEnter, OnProvinceClick, OnProvinceExit, as well as ToggleProvinceRegionSurface to color individual provinces and show ownership.

In this mode, the high-resolution frontiers should be used to ensure borders of countries and provinces match.

Option 3: Countries + provinces

This is a mixture of previous modes. Both countries and provinces are highlighted when player moves the pointer over them. When players click on a province, events are fired for both the country and the province.

In this mode, the high-resolution frontiers should be used to ensure borders of countries and provinces match.

Option 4: Cities and MountPoints

In addition to the previous 3 modes, you could add cities and mountpoints.

Each city has some interesting attributes that you can use in your game, like country and province to which belong, metropolitan population and class (country capital, regional capital or normal city).

MountPoints are created by you using the Editor or the API. They are not visible during play mode but they represent strategic locations across the map that you could use to populate your custom markers or possible destinations.

Creating your own map?

Although WMSK includes real world cartography for countries and provinces, it also provides you with the tools to create your fictionary map.

Using the Map Generator

This option included in the Map Editor component generates fully random world maps including country and province borders, city locations, names and required textures (heightmap, watermask, background texture, etc.).
Please refer to the Map Generator section in Map Editor component descriptor for more details.

Exporting and loading provinces color maps

This option does not change physical frontiers but allows you to reassign existing provinces to new countries producing fictional world maps. Using the contextual menu "Export Provinces Color Map" of the Map Editor component, World Map Strategy Kit will produce a texture of 8192x4096 pixels with all provinces in the map.

By recoloring all provinces belonging to the same country using the same color, you can create different province maps and import them at runtime using the **ImportProvincesColorMap()** method. This method will automatically generate the new countries based on the provinces colors.

Please refer to the Map Editor component for details.

Drawing your map using visual tools (Map Editor)

Please refer to the Map Editor component for details. Using the Map Editor you can:

- Start a map from scratch and manually draw the frontiers
- Import the heightmap and textures from an existing Unity terrain (optional)
- Automatically generate countries and provinces based on a texture color (Territory Importer tool)
- Edit current borders and cities and even reduce the number of countries and provinces automatically by merging existing countries or provinces.

Creating maps procedurally with code

You can also **create a map procedurally** (using scripting). Start calling **ClearAll**(), create a Country object, populate it with Regions (setting the country.regions property) and add the country to the map using **CountryAdd**. When modifying frontiers, call **OptimizeFrontiers**() and **Redraw**() to reflect the changes in map entities on the scene.

Please check demo scene 105 for code example.

The 4 representation modes in WMSK: 2D (world space), 2D (UI), 3D viewport or terrain modes.

WMSK has grown to provide a wide and rich number of options for your game presentation needs. You can build your game using one of the 4 available representations (even you could combine them in the same project). These representation modes are:

- 2D flat map in world space.
- 2D flat map as part of your UI (as a Map Panel UI element)
- 3D viewport
- Terrain mode

2D flat map only uses the main WorldMapStrategyKit prefab. It's basically a big quad that can be placed, rotated and scaled anywhere in the scene which shows a background texture according to the selected Earth style and all the map features (frontiers, latitude/longitude lines, cursor, grid, etc.).

This mode is a great choice for light-weight game UI, where a more strategic or simplistic approach is desired, like Plague Inc. or Risk. This mode will also run faster on older mobile devices. The mini-map for example uses a stand-alone map itself.

2D UI map mode uses a custom Map Panel that can be added to any Canvas UI and will render the map inside that panel. See this video for an example: https://youtu.be/QuidxfCDkiI
Also check demo scene 408 Map as UI Element under UI examples.

3D Viewport mode offers the most complete feature set including animated clouds, water, fog of war, infinite scrolling (world wrap), path-finding and functionality to position and move units across the map. It's the sandbox mode of WMSK. We recommend you use it along the advanced scenic plus styles, unless you prefer to use a simpler approach (standalone) or you need to use Unity terrain features (like trees, foliage, ground level view, …)
The viewport mode works by adding the Viewport prefab to the scene where you previously have a WorldMapStrategyKit map. The viewport will attach automatically to the WMSK object and the texture and frontiers of the map will be projected onto this viewport. See "Using the Viewport" section for more details.

You may mix the Viewport mode with the normal 2D flat map in the same scene and switch between them, see Demo scene 508 for an example.

The 3D viewport gameobject can be synced with an UI Panel so screen position / size will match. This option is useful if you want to have a 3D view of your map but limited to a certain area of the screen. To configure this mode, just drag & drop the UI Panel into the RenderViewport property of the WMSK inspector. Note that WMSK still behave as a 3D object but its position, rotation and scale are updated in realtime to match the given UI Panel location and size.

Terrain mode has been introduced in version 5. It works like the viewport mode instead of using the viewport prefab, your Unity terrain itself is assigned to the viewport property. Once you do that, the WMSK textures and frontiers will be projected onto the terrain itself using a custom terrain shader.

This mode provides more freedom since now you can use many other assets like water, sky, fog and work with gameobjects and terrain as you usually do in Unity. It will also look a lot better if you zoom and you will be able to use foliage and trees (it's a normal terrain!). Note that infinite scrolling (world-wrapping), fog of war and the animated cloud are only available in Viewport mode.

Pathfinding: Open World vs Hexagonal Grid mode (only standalone and viewport modes)

In **Open World mode**, all pathfinding functions use an internal / invisible matrix of 2048x1024 map positions to compute terrain features (land/water), elevation, blocking status and crossing cost. In this mode, no visibile grid is usually used (although it can be enabled).

In **Hexagonal Grid mode**, in addition to a visible hexagonal grid, the path finding methods are specific to the hexagonal cells. Each cell contains a terrain feature (land/water), blocking status and a different crossing cost per hexagonal edge.

There's no global switch to choose one or another mode. It just depends on the API functions you use.

Now that you have made your game design choices, continue reading this manual for a quick overview of the possibilities and functions. Then explore the demo scenes that match best your requirements and learn by example.

How to use the asset in your project

1. Import WMSK package into your project.
2. From the top menu, select GameObject -> 3D Object -> World Map Strategy Kit Map option.
3. If you want to use the 3D surface feature (also called viewport mode), also click on GameObject -> 3D Object -> World Map Strategy Kit Viewport option.

Select the WMSK GameObject created to show custom properties:

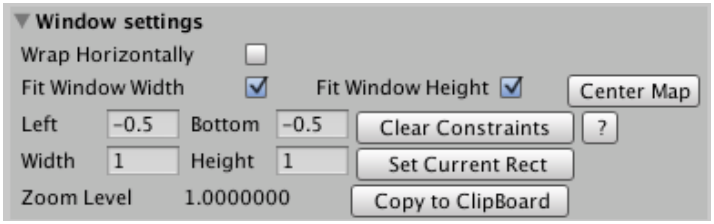
Custom Inspector Properties

Fit Width/Height/Center Map: controls and center how the map can be moved over the screen.

Rest of customization options are grouped in sections:

- **Window settings**
- **Viewport settings**
- **Earth settings**
- **Cities settings**
- **Countries settings**
- **Provinces settings**
- **Interaction settings**
- **Path Finding settings**
- **Custom Attributes settings**
- **Grid settings**
- **Miscellanea**

Window settings



- **Wrap Horizontally:** this option enables infinite horizontal scroll in viewport mode (needs to assign a viewport gameobject in viewport settings). See wrapping the world section.

- **Fit Window Width / Height:** forces map to fill current window rectangle, defined by constraints below. Click on Center Map to restore map to the center of the window rectangle.

- **Left / Bottom:** sets the position of the left/bottom corner of the map using normalized coordinates in the range of -0.5 .. 0.5 (think of the left/bottom screen corner as being -0.5, -0.5).

- **Clear Constraints:** sets current window rect to original values (left = -0.5, bottom = -0.5, width = 1, height = 1).

- **Set Current Rect**: assigns current window rect. Useful to capture current rectangle.

- **Zoom Level**: the current normalized (0..1) zoom level, used with methods like GetZoomLevel() or SetZoomLevel().

Viewport settings

- **Render Viewport:** when it has assigned a viewport gameobject, the map will show inside that viewport instead of the normal gameobject and special features will be enabled. Read “Using the Viewport feature” for more details.

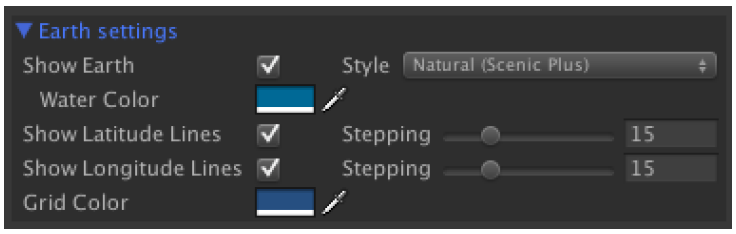
- **Ground Elevation:** sets the max. height for the surface when viewport is used.

- **Cloud Layer:** enables and customized the cloud layer when viewport is used.

- **Fog of War:** enables and customize the color of the fog of war (only when viewport is used).

- **Units Scale Multiplier**: control show units positioned on the viewport are automatically scaled. This is a scaling multiplier so when you zoom in into the viewport units can be scaled along the map. You may choose a clamp range so for instance scale of any unit can’t be less than 0.5 its original scale (or more).

Earth Settings



- **Show Earth**: shows/hide the Earth. You can for example hide the Earth and show only frontiers giving a look of futuristic UI.

- You may want to not hide the Earth, but instead use the CutOut style, which will hide the Earth, but will prevent the geographic elements and lines to be seen when they're on the back of the sphere.

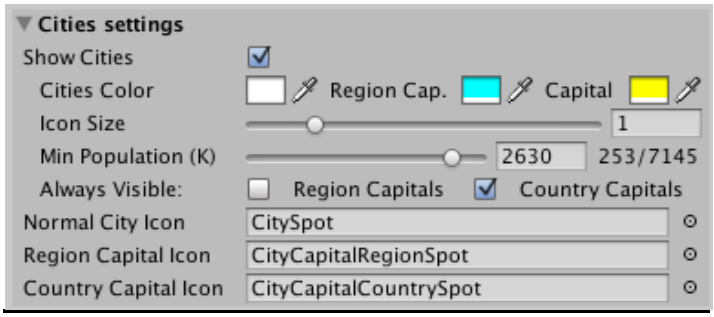
- **Style:** changes current style applied on the world map. Some styles can imply additional visual effects. See Using Scenic Styles section.

- **Water Color:** allows you to change the color of the water when Scenic styles are enabled.

- **Show Latitude/Longitude Lines:** will activate/deactivate the layers of the grid. The stepping options allow you to specify the separation in degrees between lines (for longitude is the number of lines).

- **Grid Color**: modifies the color of the material of the grid (latitude and longitude lines).

Cities Settings



- **Show Cities**: activate/deactivate the layer of cities.

- **Cities Color**: allows you to change the color of the three classes of cities included (normal cities, region and country capitals).

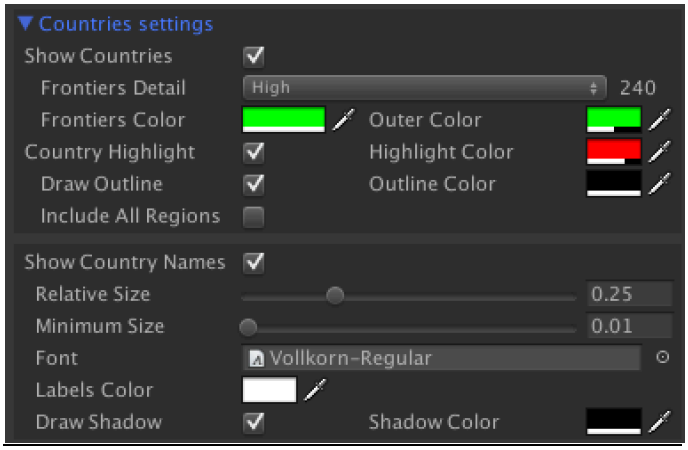
- **Icon Size**: scale multiplier for the cities icon.

- **Min Population (K)**: allows to filter cities based on metropolitan population (K = in thousands). When you move the slider to the right/left you will see the number of cities drawn below. Setting this to 0 (zero) will make all cities in the catalog visible.

- **Always Visible**: allows to ignore the minimum population filter for region or country capitals.

- **City icons**: you may override the default icons for your cities. Note that the replacement icons should be sprites (you may duplicate the city prefabs in Resources folder and assign a different sprite). If you want to show a 3D icon/game object on the location of cities, you need to populate the map using WMSK_MoveTo() method (see section "Adding your gameobjects to the viewport").

Countries Settings



- **Show Countries:** show/hide all frontiers. It applies to all countries, however you can colorize individual countries using the API.

- **Frontiers Detail:** specify the frontiers data bank in use. Low detail is the default and it's suitable for most cases (it contains definitions for frontiers at 110.000.000:1 scale). If you want to allow zoom to small regions, you may want to change to High setting (30:000:000:1 scale). Note that choosing high detail can impact performance on low-end devices.

- **Frontiers Color:** will change the color of the material used for all frontiers lines. When zooming in, the lines get a littile bit thicker and **Outer Color** is used for coloring the extra thickness of the line.

- **Country Highlight**: when activated, the countries will be highlighted when mouse hovers them. Current active country can be determined using *countryHighlighted* property (see API).

- **Highlight Color**: fill color for the highlighted country. Color of the country will revert back to the colorized color if used.

- **Draw Outline** and **Outline Color**: draws a colored border around the colorized or highlighted country.

- **Include All Regions**: when enabled, all regions of the current highligted country will be highlighted as well. If disabled (default behaviour), only the territory under the mouse will be highlighted. For instance, if you pass the mouse over USA and this option is enabled, Alaska will also be highlighted.

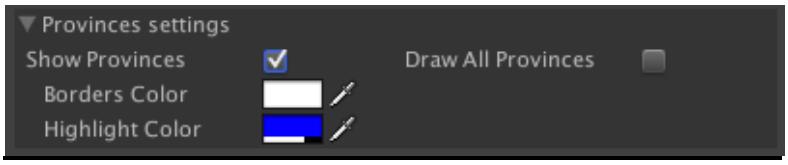
- **Show Country Names**: when enabled, country labels will be drawn and blended with the Earth map. This feature uses RenderTexture and has the following options:

- **Relative Size**: controls the amount of "fitness" for the labels. A high value will make labels grow to fill the country area.

- **Minimum Size**: specifies the minimum size for all labels. This value should be let low, so smaller areas with many countries don't overlap.
- **Font**: allows you to choose a different default font for labels (factory default is "Lato"). Note that using the decorator component you can assign individual fonts to countries.

- **Labels and shadow color**: they affect the Font material color and alpha value used for both labels and shadows. If you need to change individual label, you can get a reference to the TextMesh component of each label with Country.labelGameObject field.

Provinces Settings

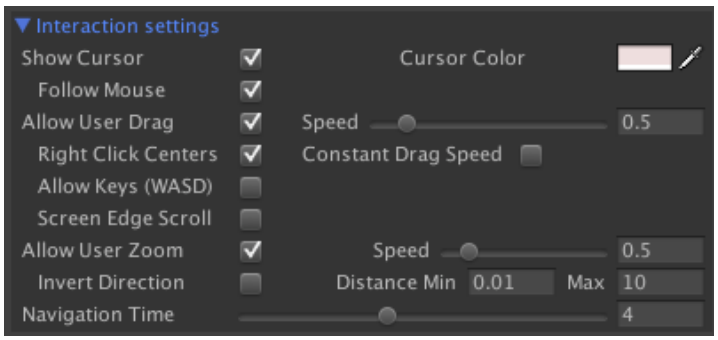


- **Show Provinces**: when enabled, individual provinces/states will be highlighted when mouse hovers them. Current active province can be determined using *provinceHighlighted* property (see API).

- **Draw All Provinces**: will render all provinces (+4100) borders on the map. Usually this toggle is unchecked what will make only provinces for currently selected country are drawn.

- **Borders Color** and **Highlight Color**: defines the color of the provinces border as well as the highlighting color (when mouse is over).

Interaction Settings



- **Show Cursor:** will display a cross centered on mouse cursor. Current location of cursor can be obtained with *cursorLocation* property when *mouseOver* property is true.

- **Cursor Color:** this is the color for the cursor cross lines.

- **Follow Mouse:** the cursor will follow the mouse position when it's over the map. If unchecked, you can change the cursor position on the map setting the *cursorLocation* property.

- **Allow User Drag:** as the title says, when enabled user can drag around the map with **Speed** parameter. You can also enable **Right Click Centers** which means that the map will scroll and center on the position the user right-clicks. **Contant Drag Speed** disables dragging acceleration **and Screen Edge Scroll** means that the map will scroll if cursor reached the edges of the screen.

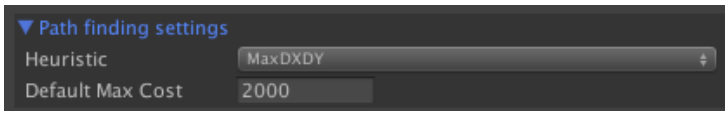
- **Allow User Zoom:** wether the user can zoom in/out the Earth with the mouse wheel.

- **Zoom Speed:** multiplying factor to the zoom in/out caused by the mouse Wheel (Allow User Zoom must be set to true for this setting to have any effect).

- **Navigation Time**: time in seconds for the fly to commands. Set it to zero to instant movements.

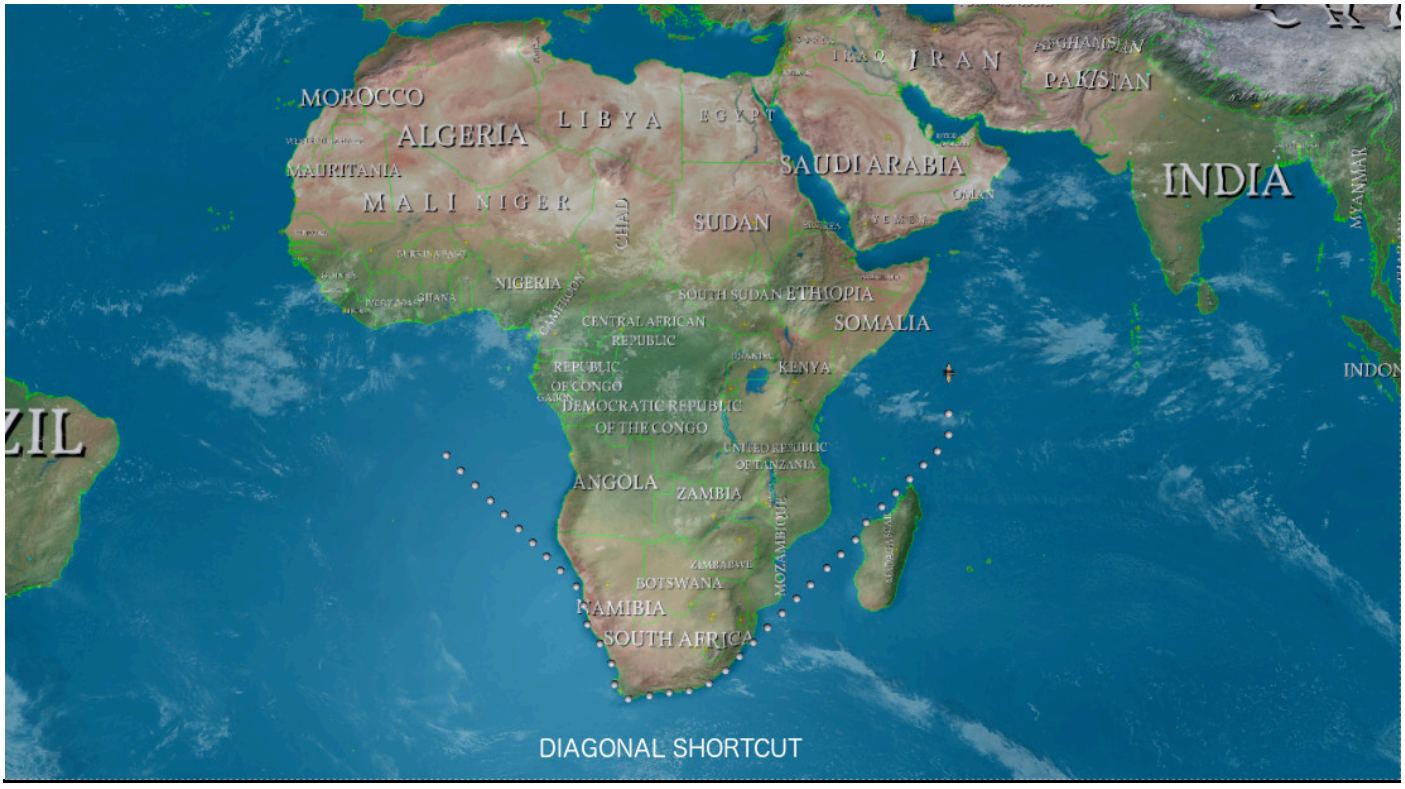
- **Enable Free Camera:** this option is only enabled in Terrain mode. When checked, main camera is allowed to be moved/rotated as usual and WMSK will sync any transform change back and forth between the internal rendering camera and the main camera

Path Finding Settings



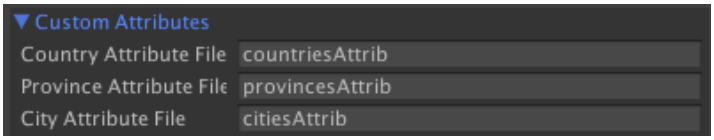
- **Heuristic:** defines the formula used when estimating the distance to the destination. It affects to the shape of the route. Below are examples of different heuristics (used in demo scene 507 “Infinite Scroll”):


MOROCCO
ALGERIA
LIBYA
EGYPT
IRAN
PAKISTAN
SAUDI ARABIA
INDIA
MAURITANIA
MALI
NIGER
CHAD
SUDAN
NIGERIA
SOUTH SUDAN
ETHIOPIA
SOMALIA
CENTRAL AFRICAN REPUBLIC
KENYA
REPUBLIC OF CONGO
DEMOCRATIC REPUBLIC OF THE CONGO
ANGOLA
ZAMBIA
BOTSWANA
NAMIBIA
SOUTH AFRICA
MAXDXDY


MOROCCO
ALGERIA
LIBYA
EGYPT
IRAN
PAKISTAN
SAUDI ARABIA
INDIA
MAURITANIA
MALI
NIGER
CHAD
SUDAN
NIGERIA
SOUTH SUDAN
ETHIOPIA
SOMALIA
CENTRAL AFRICAN REPUBLIC
KENYA
REPUBLIC OF CONGO
DEMOCRATIC REPUBLIC OF THE CONGO
ANGOLA
ZAMBIA
BOTSWANA
NAMIBIA
SOUTH AFRICA
DIAGONAL SHORTCUT

- **Default Max Cost**: this setting defined the maximum allowed cost of the route, meaning 1 point per horizontal/vertical movement and 2.64 points per diagonals. Use a big number if you want to route across large areas of the world (default value is 200000) or reduce this value to increase performance and limit routes to nearer zones.

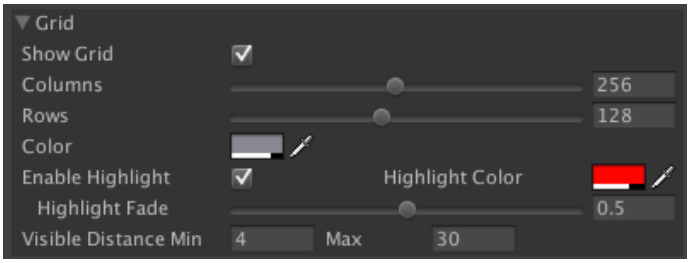
Custom Attributes Settings



In this section you can change default filenames for countries, provinces and cities custom attributes files. These files are stored in Resources/WMSK/Geodata folder.

You can change these names to use different attribute sets.

Grid Settings



- **Show Grid:** enables/disables hexagonal grid. When set for first time, the cells list and mesh is created and rendered according to the following settings.
- **Columns:** number of columns in the grid.
- **Rows:** number of rows in the grid. Usually this value will be the half of columns.
- **Color**: the color of the hexagonal grid. You can use different alpha values for transparent grid.
- **Enable Highlight**: if cells will be highlighted when mouse hovers them.
- **Highlight Color / Fade**: speficies the color and fading effect to the currently highlighted cell.
- **Visible Distance (Min / Max)**: distance in world units from the Camera to the map where the grid is visible. Outside of this range, the grid will fade out gracefully until it gets invisible.

Miscellanea

- **Prewarm At Start:** when enabled, the asset will perform some heavy computation during initialization to prevent hiccups during play. Some of these computations are the highlighting of big/complex countries (Russia, Antarctica, Canada, Greenland) and the navigation matrices of PathFinding engine.

- **Geodata Folder Prefix:** in case you modify the geodata files provided with the asset (for example if you want to modify frontiers, add new countries or cities, etc.) then you will want to use a different folder for the modified files (so when the asset is updated you don't need to backup/restore the geodata folder). In this case, you need to specify here the location of your geodata folder.

Choose Reset option from the gear icon to revert values to factory defaults.

Using the Viewport mode feature

The asset allows to render the map inside a Viewport game object (provided in the asset as a Prefab).

This mode adds the following enhancements vs the normal 2D/flat mode:

- Allows **cropping** the map inside the rectangle defined by the viewport gameobject area when panning or zoomin in.
- Renders the Earth surface over a **3D mesh with real elevation based on heightmap**.
- Can add **world space gameobjects over the 3D surface** and below the cloud layer. It can hide automatically game objects positioned on top of the map, and control their scale and
- Can add a **fog of war** layer that can obscure the gameobjects positioned on the map as well as the map itself.
- Enables the **cloud layer** which is rendered on top of fog of war and the game objects, with **drop shadows over the map**.

We recommend using the viewport feature to take advantage of all the above features for your game, as it makes it more visual appealing. However, mind that the viewport uses a RenderTexture of 2048x1024 pixels in size and also it uses additional memory and CPU to compute the 3D surface mesh in real-time. **Not all mobile devices may support this feature. Try the provided demos on your devices to test if they can work with it.**

To use the viewport feature:

1- Drag the new Viewport prefab to the scene (from Resources/Prefabs folder).
2- Assign the new viewport gameobject in the scene to the viewport property of the WorldMap in the inspector (you can also do that using code, check the demo.cs script for example)
3- That's all! The map will show up inside the viewport.

To deactivate the viewport:

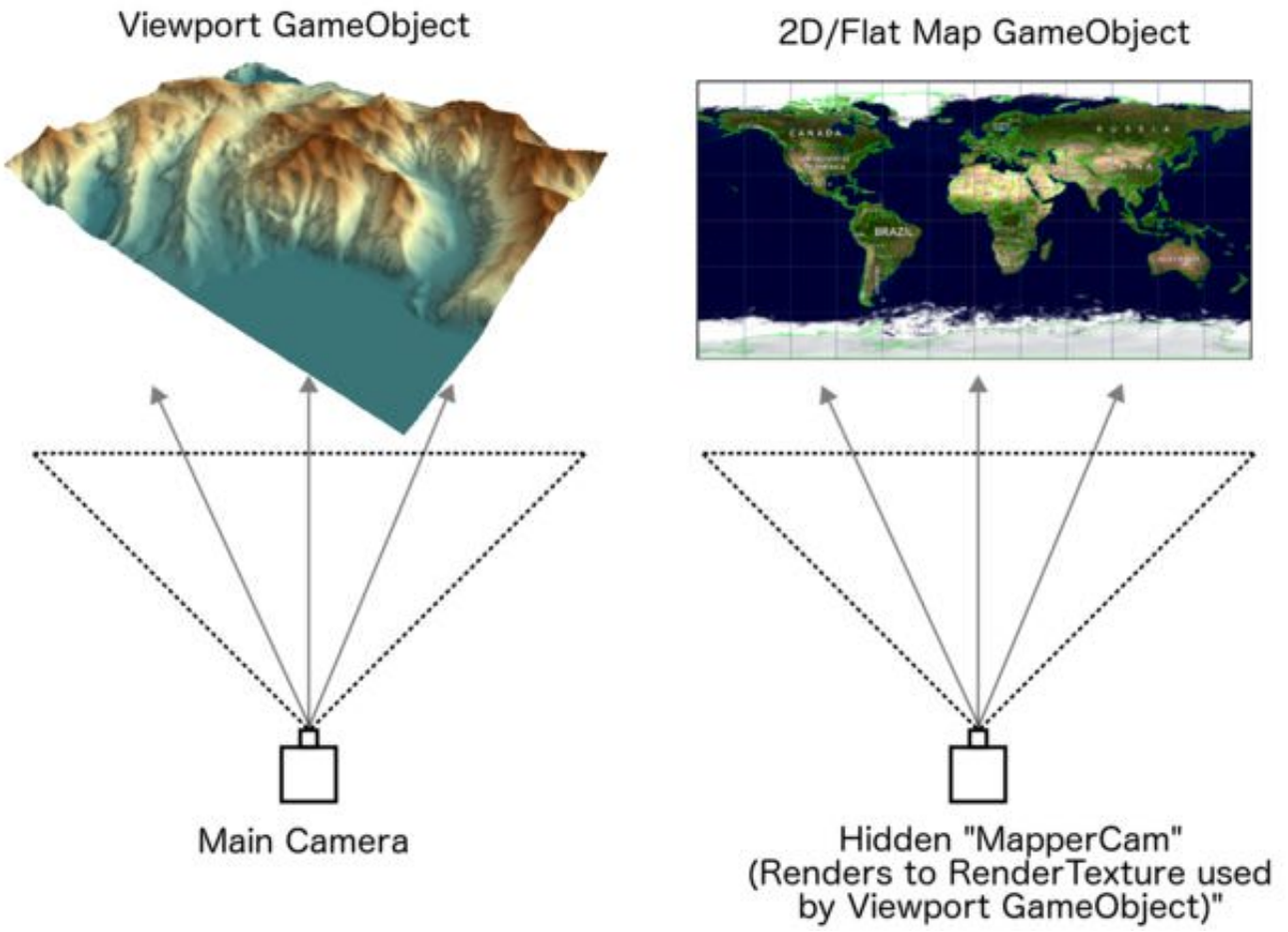
1- Select the viewport field in the inspector of WorldMap component and click delete (basically, remove the reference to the viewport game object).
2- Delete the viewport game object from the scene.

UI Mode and Viewport

When a viewport is in the scene, a UI Panel property will show up in the WMSK inspector. By assigning an UI Panel to this property, WMSK will sync its position, rotation and scale to match the UI Panel position and size at any moment.

How does Viewport mode work?

When you use the viewport mode, the following setup will be contained in your scene (yes, there're 2 cameras):



In viewport mode, there're 2 key gameobjects in the scene. The viewport and the normal 2D/Flat map. The 2D/Flat map is positioned out of the Main Camera frustum so it's not visible. A hidden camera created by WMSK named "MapperCam" captures the image from the normal map and renders it to a RenderTexture which is used by the Viewport gameobject's material.

So in viewport mode, when you zoom in/out, it's the hidden "MapperCam" which moves towards or away from the normal 2D map. Same when you drag the mouse over the viewport, those gestures are captured by WMSK and converted into translations to the hidden camera which moves horizontally with respect the normal map.

Now, there're two ways of adding objects to the maps.

- 3D objects, like units that should reflect "height" or 3D aspect, can be added calling WMSK_MoveTo(position). When you call this method as it's, the gameobject receives a GameObjectAnimator component that takes care of syncronizing its world space position and scale to match the current view on the viewport. When you drag or zoom, those units really move around in world space. They are not parented to the viewport but placed on top of the viewport in world space.

- 2D objects, like country names, lines, circles or country textures/meshes, are added on top of the normal 2D map gameobject. They are captured by the MapperCam camera and are simply blended with the

background map texture and result in the material texture used by the viewport. That's how country names adapt to the 3D shape or mountains shown on the viewport, because they're just part of the texture. The following methods add objects to the normal map: **AddLine** (with arcElevation parameter=0), **AddCircle**, **AddMarker2DSprite**, **AddMarker2DText**, **Marker3DObject**, **ToggleCountrySurface** or **ToggleProvinceSurface** and similar methods. Those objects do not receive a GameObjectAnimator, they stay on their positions.

So, if you want to add static sprites (or control their positions with your own code and not GameObjectAnimator) you can use the AddMarker2DSprite method. This method accepts an "enableEvents" parameter that enables any script attached to your sprite to receive the OnMarkerClick event. WMSK will automatically add a *MarkerClickHandler* script to your sprite gameobject so clicking and dragging is possible in an automatic way:

```
MarkerClickHandler handler = map.AddMarker2DSprite(mySpriteGameObject, position, scale, enableEvents: true, autoScale: true);
handler.allowDrag = true; (by default = true; allows dragging the sprite with the mouse)
handler.captureClickEvents = true; (by default = true; prevents clicks going through the sprite to the map).
```

The MarkerClickHandler also exposes a few events that you can use to detect when a drag starts or a click is performed over the marker for example.

Adding game objects to the viewport (demo scene 503)

When you enable the viewport mode, you can make your game objects be part of the viewport itself so it will take control of its position, orientation, scale and visibility. When you scroll the viewport across the map, Game Objects positioned this way will automatically disappear when they get off the viewport screen region. This is different from adding a Game Object to the map itself (using Markers API) because your Game Objects will preserve its 3D appearance whereas adding them with the Markers API will make them appear flat in the viewport (as part of the texture).

WMSK includes a set of APIs designed to make it very easy to add your Game Objects to the viewport.

When the asset is imported in your project, a few extensions are automatically added to the GameObject class. These extensions are defined in the script WMSKGameObjectExtensions.cs.

To add or move a game object named “GO” across the map you can use:

GO.**WMSK_MoveTo(float x, float y, float duration, DURATION_TYPE durationType);**

This will make the gameobject part of the game map and will move it from current position to the map coordinates (x,y) with duration in seconds. Usually, you will call this method to create/add your graphic elements to the map the first time with a 0 as duration.
As per the map coordinates, both x and y ranges from -0.5 to 0.5, being 0,0, the center of the World Map.
The parameter “durationType” specified how to apply the given duration. It can be:

- “Step”: each movement step will take the same duration in seconds.
- “Route”: the complete movement will take the given duration in seconds.
- “MapLap”: the duration is specified in terms of seconds taken to cross the entire map. This is useful to achieve a consistent movement speed if you pass a custom list of points that are not continuous.

Check out demo scene Map Population's code used to move the tank around for an example.
GO.**WMSK_MoveTo(Vector2 destination, float duration, DURATION_TYPE durationType);**

Same than previous function except it accepts a Vector2 parameter.

GO.**WMSK_MoveTo(Vector2 destination, float duration, DURATION_TYPE durationType, bool scaleOnZoom, float altitude);**

This method will add the gameobject *GO* to the destination in *duration* seconds and also position it at *height* meters from the ground according to the heightOffsetMode which can accept the following values:

- Absolute_Height: position the unit at an absolute height position. Useful for marking heights.
- Absolute_Clamping: position the unit at an absolute height position BUT never below ground level. Useful for aerial units.
- Relative_To_Ground: simply adds height to the ground altitude at that point. Useful for land units or land-marking.

ScaleOnZoom will make the gameobject scale change as the camera zooms in/out. If you want to preserve the gameobject size on screen regardless of the current map zoom, pass *false* to this parameter.

GO.**WMSK_MakeChild();**

This is a convenient shortcut to "WMSK_MoveTo(Vector2.zero, 0, false)". What it does is parent the given gameobject to the map, placing it on the center of the map and preserving its localscale. This is useful if you want to have a common invisible placeholder in the hierarchy for several game objects on the map, so you can for example remove all of them at once just deleting the parent placeholder. Check DemoInfiniteScroll demo scene for an example (the ship trail uses this approach).

GO.**WMSK_GetMap2DPosition();**

This method will return a Vector2 object with the map coordinates of the game object in the range (-0.5, 0.5).

GO.**WMSK_FindRoute(Vector2 destination);**

This method will return a list of Vector2 map coordinates corresponding to the optimal route of the GO from current position to destination, having into account its terrain capabilities.

GO.**WMSK_LookAt(Vector2 destination);**

Makes this unit rotate and point towards destination.

GO.**WMSK_Fire(GameObject bullet, Vector3 anchor, Vector2 mapDestination, float duration, float shootArc);**

This method will fire the bullet GameObject starting at "anchor" and aimed to mapDestination for a duration and also following an arc with altitute of "shootArc".

The anchor is the relative position to the center of the firing unit. Anchor is defined in local scale coordinates, assuming a reference size of 1x1x1.

See demo scene 510 for an example.

Customizing units

The **WMSK_MoveTo** method will return a reference to a **GameObjectAnimator** component which will be attached to your game object. This component is the link between your gameobject and the asset itself. We recommend you to take a look at the code sample behind the MapPopulation demo.

The GameObjectAnimator component contains some useful properties that allows more precise control over the movement of the game object over the map. The following properties can be changed although they will use default values if not used:

- **uniqueId**: optional user-defined integer that identifies uniquely this game object.
- **type**: this is an integer, a user-defined value, that can be used to identify the type of unit.
- **group**: this is an integer, a user-defined value, that can be used to group units.
- **player**: this is an integer, a user-defined value, that can refer to the player to which the unit belongs to.
- **duration**: the duration of the current move.
- **easeType**: type of easing for the movement: it can be one of the following types: EaseIn, EaseOut, Exponential, Linear, Smooth and Smoother.
- **height**: the current height from the ground.
- **heightMode**: changed the height parameter semantic. Accepts Absolute_Height, Absolute_Clamped (never position below ground level) or Relative_To_Ground.
- **autoScale**: toogles changing the scale automatically when user zooms in/out.
- **follow**: toggles camera following during current movement.
- **followZoomLevel**: the zoom level for the camera follow.
- **autoRotation**: set it to true to make the game object rotate automatically then routing to the destination as well as adapt to the ground. Set this to false for static objects, like buildings (default value).
- **rotationSpeed**: the speed factor for the autorotation property.
- **preserveOriginalRotation**: when set to true, game object will retain its current rotation when added to the viewport. Note that autoRotation will override this property so if autoRotation = true, preserveOriginalRotation will be ignored.
- **terrainCapability**: sets whether the game object can pass through only water, only ground or any terrain (default is Any). This is an important property of your game object and if set to a value different than Any, the game object will not move in a straight line but along a calculated route from current position to end position.
- **enableBuoyancyEffect:** if this unit can be affected by buoyancy effects (subtle animation due to sea waves). Is true by default.
- **pivotY:** specifies the Y position of the pivot (0..1). If the model is designed so the pivot is at bottom the you don't need to specify this value. Most ground units should be designed this way so when you put a tank unit for instance over the ground at (0,0,0), the tank will appear entirely over the ground. However, if your model is not designed this way, you can specify a value for pivotY. A 0.5 value will mean that the pivot is at the center of the mesh, while a value of 1 means that the pivot is at the top of the mesh.
- **minAltitude/maxAltitude**: these values (0..1) define the altitude range across the game object can move. Default values are 0 to minAltutude and 1 to maxAltitude, so there's no limitation. You may want to lower the maxAltitude to prevent ground-level game objects cross very high mountains.
- **maxSearchCost:** the maximum allowed cost for the route. A value of -1 will use the global setting defined by pathFindingMaxCost.

- **maxSearchSteps:** the maximum allowed steps for the route. A value of -1 will use the global setting defined by pathFindingMaxSteps.
- **attrib**: JSON object that stores user-defined Custom Attributes (see Custom Attributes section for more details)
- **BlockRayCast:** when set to true, other map click events won't be fired when the pointer is on this unit.
- **visible:** (default value=true). When set to false, unit will be invisible regardless of its position on the screen or map. For a visible unit, use isVisibleInViewport to determine if the unit is currently visible on the screen.
- **updateWhenOffScreen:** (default value=false). By default, units not visible in the viewport are not updated in world space (position/rotation is not updated to improve performance when using many units). Set this value to true to keep units updating their transform even when they're out of the viewport. Note that when units move, their GameObjectAnimator properties still are updated regardless of their visibility (for example, units moving will have their currentMap2DLocation field updated).

The GameObjectAnimator also exposes a few useful direct functions that you can call once you have a GameObjectAnimator reference to interrogate the unit status or to send direct commands to them:

- **isOnWater:** returns true if the unit is currently on water (always true for naval units and false for ground units). Some units can move either over ground and water (air and hybrid units) and should have terrainCapability set to Any.
- **isNear:** returns true if the unit is near given coordinate (optionally passing a max distance value).
- **isVisibleInViewport**: returns true when the gameobject is visible in the current viewport view. Remember that you can zoom in and scroll the viewport, so the asset will automatically hide game objects that fall outside the current view.
- **MoveTo:** similar to WMSK_MoveTo() method extensions. In fact, WMSK_MoveTo() calls this function internally. WMSK_* methods are available for convenience if you don't have a reference to the GameObjectAnimator (which you can get using GameObjectAnimator anim = yourUnit.GetComponent<GameObjectAnimator>()).
- **FindRoute:** same than WMSK_FindRoute.
- **isMoving**: returns true if the gameobject is currently moving across the map.
- **ChangeDuration**: adds/substract time to the path duration while unit is moving effectively changing the unit speed.
- **currentMap2DLocation**: the current location of the game object in map coordinates (-0.5 .. 0.5).
- **endingMap2DLocation**: the final location of the game object in the map when it finish entire movement.
- **destination**: the destination ff the current movement. If the unit moves along a path, destination marks the end position of current step. If the unit does not follow a path (ie. called with MoveTo(destination), then destination equals to endingMap2DLocation).
- **Stop**: makes the unit stop at current position.
- **Fire**: similar to WMSK_Fire() method extension.
- **GetCellNeighbours**: gets a list of hexagonal cells where the unit can move (if grid is enabled).

In addition to the above properties and methods, each GameObjectAnimator exposes the following events which you may subscribe by adding your own function to this properties (eg. OnMoveStart += myfunction):

- **OnMoveStart(GameObjectAnimator):** fired when the game object starts a movement.
- **OnMove(GameObjectAnimator):** fired when the game object moves over the map.

- **OnMoveEnd(GameObjectAnimator):** fired when the game object stops.
- **OnCountryEnter(GameObjectAnimator)**: fired when the unit enters a country.
- **OnCountryRegionEnter(GameObjectAnimator)**: fired when the unit enters a country region.
- **OnProvinceEnter(GameObjectAnimator)**: fired when the unit enters a province.
- **OnProvinceRegionEnter(GameObjectAnimator)**: fired when the unit enters a province region.
- **OnPointerXXXX:** see Unit selection events on next sections.
- **OnKilled:** fired when unit gameObject is destroyed

Useful general APIs for GameObjectAnimators

map.VGOToggleGroupVisibility(group, visible): toggles visibility of a group of registered game objects in the viewport.

map.VGOGet(uniqueId): returns the registered game object in the viewport by its unique identifier (optionally set as a property).

map.VGOGet(position, distance): returns the registered game object in the viewport located near position.

map.VGOGet(rect): returns a list of registered game objects contained in a given rect.

map.VGOGet(rect, List<GameObjectAnimator> results, AttribPredicate predicate): returns a list of registered game objects contained in a given rect which optionally satisfy a predicate function.

map.VGOGet(List<GameObjectAnimator> gos): fills user supplied list with all registered gameobjects.

map.VGOBuoyancyAmplitude: rotation amount for the buoyancy effect of naval units. Set to zero to deactivate.

map.VGOBuoyancyMaxZoomLevel: maximum zoom level where the buoyancy effect is enabled.

map.VGOLastClicked: returns last clicked unit.

Unit selection events (demo scene 509)

When you call WMSK_MoveTo() method to move an unit to the viewport, it received a BoxCollider component if it doesn't already contain their own collider (this collider component is responsible for triggering internal events produced by mouse hovering and clicking on them – if the collider is destroyed, the unit will be no longer selectable).

The following 4 events exposed by the GameObjectAnimator component can be used to detect mouse events:
OnPointerEnter: triggered when the mouse enters the gameobject.
OnPointerExit: triggered when the mouse abandon the gameobject.
OnPointerDown: triggered once when mouse left button is pressed over the gameobject.
OpPointerUp: triggered when the left mouse left button is released on the gameobject.
OnPointerRightDown: triggered once when mouse right button is pressed over the gameobject.
OpPointerRightUp: triggered when the mouse right button is released on the gameobject.

For example, if you want to receive events from a specific unit "tank1" you do:
tank1.OnPointerEnter += (GameObjectAnimator anim) => Debug.Log ("Tank1 mouse enter event.");

The above events are useful at individual unit level, but there're also 4 global actions that you can use to detect mouse interactions with any unit. These global actions are defined in WMSKViewportGameObject.cs:

Action<GameObjectAnimator> OnVGOPointerEnter; (Mouse enters the gameobject)
Action<GameObjectAnimator> OnVGOPointerExit; (Mouse abandon the gameobject)
Action<GameObjectAnimator> OnVGOPointerDown; (Left mouse button pressed on the gameobject)
Action<GameObjectAnimator> OnVGOPointerUp; (Left mouse button released on the gameobject)
Action<GameObjectAnimator> OnVGOPointerRightDown; (Right button pressed the gameobject)
Action<GameObjectAnimator> OnVGOPointerRightUp; (Right button released on the gameobject)

You can assign your own delegate method to these actions and it will receive the reference to the GameObjectAnimator that generated the mouse event. This way you can have a single point or global approach to mouse event handling for all units. Remember that the GameObjectAnimator is nothing more than a component attached to the unit gameobject.

For example if you want to receive a mouse event for **any** unit you do:

```
map.OnVGOPointerDown = delegate(GameObjectAnimator obj) {
      Debug.Log ("Mouse button pressed on " + obj.name);
};
```

Selecting multiple units with a rectangle selection (Demo scene 509)

Starting version 2.3 WMSK provides an useful API which you may call to initiate a rectangular selection. Just call:

map.RectangleSelectionInitiate(callback, rectangleColor, rectangleLineColor);

Once you call this function, the user will no longer be able to drag round the map but still can make a rectangle selection, clicking and dragging over an area.

Each time the rectangle size changes, the callback function is called passing the Rect coordinates and a boolean indicating if the mouse button has been released hence the selection has ended.

You can customize the rectangle selection passing a fill color and a color for the animated border line.

map.rectangleSelectionInProgress returns true if a rectangle selection is occuring.

map.RectangleSelectionCancel will abort current rectangle selection.

Path Finding (non grid based)

WMSK provides 3 path finding methods that don't require an hexagonal grid (for hexagonal grid-based pathfinding see next section):

Open world movement

In this mode, any path can be computed between two points on the map. This mode uses an invisible matrix cost of 2048x1024 positions mapped to the world plane. The ability to cross a point is determined by the unit terrain capability and the crossing cost at each position of this matrix.

Using Path Finding feature with Game Objects added to the viewport (demo scene 201 in Path Finding Examples folder)

When you set the terrain capability of any unit (using terrainCapability property of Game Object Animator component returned after you add the game object with WMSK_MoveTo()) to a value different than ANY, the engine will automatically perform a path search each time you move the unit to a destination position (using again WMSK_MoveTo()).

The path finding engine works by having two precomputed cost matrices: one basic matrix for identifying ground and water positions and another one to store user-defined costs.

The first matrix is automatically computed the first time the engine searches for a path (or when the prewarm option is enabled during the startup of the asset). So you don't have to worry about this one. This matrix takes the heightmap and water mask of the scenic style to determine which areas are suitable for water or ground units.

The second matrix allows you to specify custom movement costs on the map. This matrix is accessed through **PathFindingCustomRouteMatrix** property. Note that each time you use this property a copy of the current matrix is returned. You can use this property to change set the entire cost array at once, for example when player turn changes or when units have different map costs before issuing a MoveTo() call to them.

You would want to specify custom costs because:

- Some provinces might be blockaded/blocked off by enemy units.
- Some provinces might have a fortress in the way
- Some terrain types may be impassable during certain weather conditions on the map
- Enemy or neutral provinces might be in the way and need to be dealt with
- Units may need to choose the fastest route, taking into account railway lines and road levels (via attributes in the provinces)
- A nuke could have been dropped in a province making it off-limits for x number of turns
- Country X may not have given country Y the right to transport troops through their provinces
- Diplomatic agreement
- …

The format of the second matrix (the one returned by **PathFindingCustomRouteMatrix**) is a linearized 2D array of integers (an int[] in C#). The size of this matrix equals to 2048x1024 positions. On each position of the matrix, the integer value could be:

- 0 (zero): meaning that position is unbreakable.

- -1: the cost of that position has not yet determined and will be returned on the fly by an event function (see below).
- 1 or more: the custom cost for crossing through this cell.

Check out the different **PathFindingCustomRouteMatrixSet** functions to fill regions of the matrix with your own values. The demo scene 201 ("Path Finding Advanced") contains sample code where you can see how to specify the onwership of each country and set this custom route matrix accordingly, so the units can or can't pass through certain countries.

It's also possible to set the event **OnPathFindingCrossPosition** so it will be fired when the path finding engine needs to know the cost for certain location (when the value for the matrix at that position is -1), instead of pre-populating the matrix with **PathFindingCustomRouteMatrixSet** functions. However, it's more efficient to use the later set of functions if you need to assign the same cost for a country, a province or a set of regions.

Please refer to the Demo Scenes MapPopulation (demo #503), PathAndLines (demo #504) and PathFindingAdvanced (demo #201) for sample code.

Manually getting paths between two map points (non-grid based)

Simply call **map.FindRoute** API. This function has several overloads. It can accept two map points and optional movement parameters, like terrain capability, altitude range and max search cost. It returns null if no path is found, or a list of Vector2 values that corresponds to map positions.

For game objects positioned on the viewport, you can just call **go.FindRoute**(destinationPoint) where go is the gameObject, and it will return the path having into account the unit movement properties, like the terrain capability.

Country to Country path finding (demo scene 202 in Path Finding Examples folder)

WMSK also allows you to determine the optimal connections between 2 countries in the world. The APIs to use are:
FindRoute(startingCountry, destinationCountry) will return the list of country indices between the two countries, including them. You can use **country.canCross** to determine if the route can pass through any given country. Also **country.crossCost** allows you to specify a custom cost for crossing that country. And finally each country object has a **neighbours** array of boundary countries which you may modify to define custom connections between countries.

Province to Province path finding (demo scene 203 in Path Finding Examples folder)

Similar to the previous method but using Provinces. The APIs to use are:
FindRoute(startingProvince, destinationProvince) will return the list of province indices between the two provinces, including them. You can use **province.canCross** to determine if the route can pass through any given province. Also **province.crossCost** allows you to specify a custom cost for crossing that province. And finally each province object has a **neighbours** array of boundary provinces which you may modify to define custom connections between provinces.

Path Finding (grid based)

Hexagonal grid paths (demo scenes 204-205 in Path Finding Examples folder)

In this case, the hexagonal grid is used as a pattern for the potential paths. Each cell stores costs for crossing hexagonal edges as well as blocking status or ground/water status.

WMSK uses an internal array called cellCosts which is accessible through **pathFindingCustomCellCosts** property. You can use this property to manually get or set a different cost array if you need to quickly swap between different configurations.

Useful APIs:

FindRoute(startingCell, destinationCell) will return the list of cell indices between the two cells, including them. You can use **cell.canCross** to determine if the route can pass through any given cell.

PathFindingCellSetSideCost(cell, side, cost): assigns a crossing cost for a cell and a specific side of the hexagon. Note that the cost is applied when moving outside the cell. If you want to set the same cost for a side irrespective of the direction of movement, use the other overloaded method: **PathFindingCellSetSideCost (cell1, cell2, cost)**.

PathFindingCellSetAllSidesCost(cell, cost): assigns the same crossing cost for all sides of the hexagonal cell.

PathFindingCellSetSideCost(cell1, cell2, cost): assigns a crossing cost between cell1 and cell2. The proper side is automatically determined.

PathFindingCellGetSideCost(cell1, side): gets current crossing cost for that cell's side.

For gameObjects positioned on the viewport, you can also call **go.FindRoute(destinationCell, …)**.

Pausing the game

Sometimes you need to pause the game (for instance to show an in-game menu). You can pause the game using the paused property. Example:

WMSK.instance.paused = true;

Also, you can control the overall game speed at which time flows using the TImeSpeed property (which defaults to 1):

WMSK.instance.timeSpeed = 1.25f;

Wrapping the world (demo scene 507)

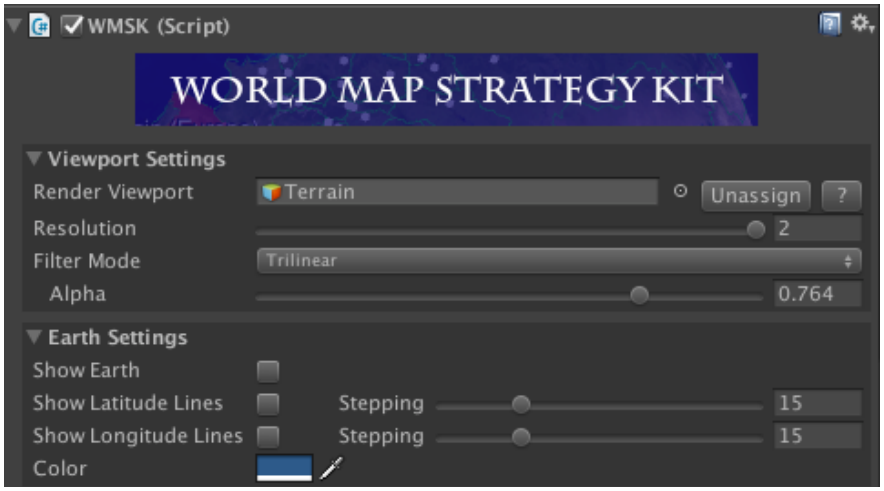
Starting V1.4 you can enable infinite horizontal scrolling. Just tick the “Wrap horizontally” checkbox in Windows settings.

When enabled, path finding engine will take into account the minimum distance across borders to choose the optimal route. Naval units will seamlessly move from Alaska to Russia coasts. Hexagonal grid will also draw seamlessly around the world. Aerial trajectories will also work.

There’s currently one limitation regarding ground-level or non-elevated lines, which are still not compatible with wrapping mode. This means that flat lines will be drawn as if no wrapping occurs. A workaround for this would be to add a small elevation amount, so WMSK will use an alternate LineRenderer which is indeed compatible with wrapping mode. Dashed lines and elevated lines use this alternate line renderer and they works in this mode.

Using the Terrain mode

Starting version 5, WMSK allows you to use the Unity standard terrain itself into the Viewport slot:



When you assign your terrain to the "Render Viewport" slot, the WMSK gameobject will be moved away and controlled in a special way. Also your terrain will get a custom shader so it can blend the WMSK features along with the terrain splatmap textures.

The Resolution option allows you to adjust the resolution of the internal render texture used to capture WMSK's features and imprint them onto the Unity terrain.

Filter Mode let's you set the texture filtering option.

Alpha setting specifies the transparency of the WMSK's features visible on the terrain map.

If you want to show the terrain splatmap textures instead of the Earth background texture, unselect "Show Earth" and increase the Alpha value. Once you do this, the country frontiers will be blended with the splatmap textures!

Check out the Terrain demos for examples (Demos/Terrain folder).

Modifying frontiers at runtime

WMSK includes a new API for transfering a region from country B to another country A modifying both country frontiers at runtime. This could be seen as if country A conquered part or all country B (remember: a country can have more than one land region – for example, a country with a mainland and 2 islands will have 3 regions in total).

The list of regions of a country can be obtained through the *regions* list of the country object. The main region of a country is defined by the biggest region and is identified by the *mainRegionIndex* of the country object. So *country.regions[country.mainRegionIndex]* would return the reference to the main land region for the country.

For example, to transfer the main region of Portugal to Spain you could do:

```
Country portugal = map.GetCountry("Portugal");
Region portugalMainRegion = portugal.regions[portugal.mainRegionindex];
int spainCountryIndex = map.GetCountryIndex("Spain");
map.CountryTransferCountryRegion(spainCountryIndex, portugalMainRegion);
map.Redraw();
```

Check demo scene 102 for an example of how to grow an empire using this API.

In WMSK 2.3, Provinces can also be merged at runtime. Check demo scene 103 for an example.

In WMSK 2.5, a province can be converted to an independent country using ProvinceToCountry method.

Limitations

It's important to note that the above API is experimental and currently shows some important limitations:

- It's quite slow with high definition frontiers, specially with bigger countries, like Canada or Russia.
- It has only been tested with country low resolution frontiers.

These limitations derive from the fact that automating boolean operations on complex polygons result in a extremmely difficult task so it's not 100% reliable. We provide this new API in beta – after extensive tests we find that this API could be useful as is with the world map provided taking into account above limitations but again you should make your own tests if you modify the map.

Using the Scenic Styles

In addition to classic textured styles for the map, there're several advanced "Scenic Styles" included: "Scenic", "Scenic Plus" and "Scenic Plus Alternate 1". When enabled, the asset will use custom shaders to provide special effects like bump mapping, clouds and water animations.

- **Scenic style** uses textures of up to 2K resolution. This style adds bump mapping and clouds plus simulated cloud shadows effect of the ground.

- **Scenic Plus** uses textures of up to 8K resolution and adds water animation and coast foam. It will also fade the high-resolution texture to a diffused texture when zooming in to prevent excessive pixellation. Scenic Plus is itended to work with viewport with the cloud layer enabled.

- **Scenic Plus Alternate 1** is a Scenic Plus variant which uses a different texture for the Earth and won't fade into a blurred texture when you zoom in. Therefore, this shader variant is a bit more efficient than the Scenic Plus.

Scenic Plus styles offer additional visualization options in the inspector that allows you to customize the water level and foam effect.

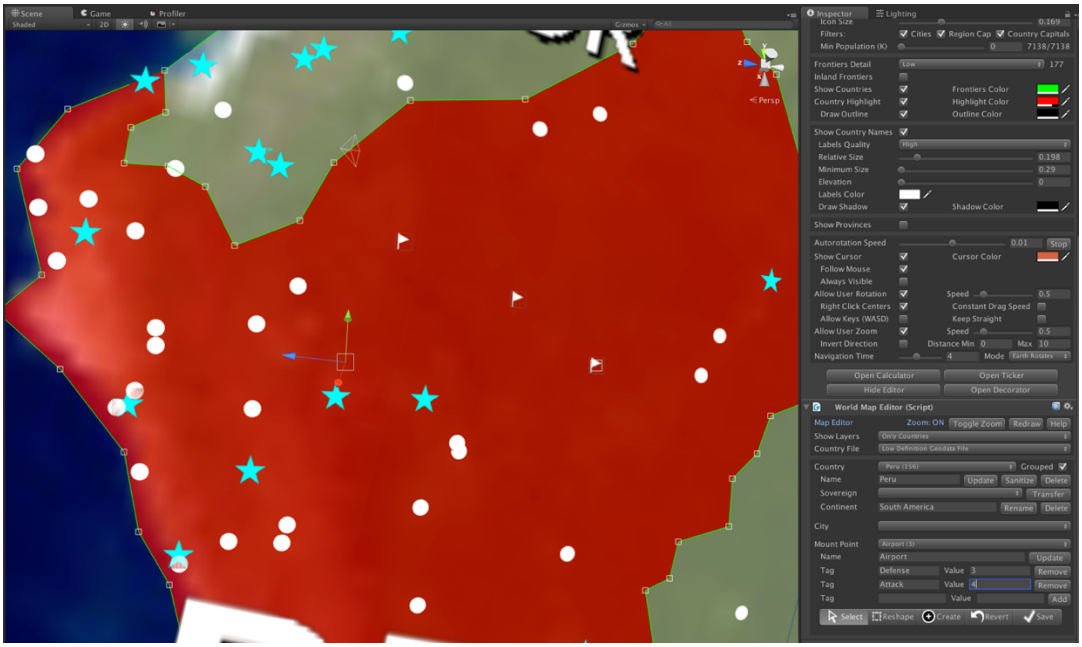
Mount Points

Mount Points are user-defined markers on the map created in the Map Editor. Basically, a mount point is a special location that includes a name, a class identifier (an user-defined number) and a collection of tags.

Mount Points are useful to add user-defined strategic locations, like airports, military units, resources and other landmarks useful for your application or game. To better describe your mount points, WPM allows you to define any number of tags (or attributes) per mount point. The list of tags is implemented as a dictionary of strings pairs, so you can assign each mount point information like ("Defense", "3") and ("Attack", "2"), or ("Capacity", "10"), ("Mineral", "Uranium") and so on.

Note that Mount Points are invisible during play mode since they are only placeholder for your game objects. The list of mount points is accesible through the mountPoints property of the map API.

Mount Points appear during design time (not in playmode) as a flag:



The editor provides a **Mount Point Mass Creation Tool**, so you can quickly populate countries, provinces and entire continents with random mount points!

Custom Attributes (demo scene 101)

Starting V1.2 you can extend countries, provinces and cities metadata with your own set of attribures (same as Mount Points). We call this feature "Custom Attributes".

Custom Attributes are stored in separated files in the same Geodata folder which contains countries, provicnes and cities borders data. The default file names are "**countryAttrib"**, "**provincesAttrib"** and "**citiesAttrib"**.

Assigning and retrieving your own attributes

Demo scene #101 covers all uses cases regarding Custom Attributes. For example, to add a few attributes to a country, like Canada you would do:

```
Country canada = map.GetCountry("Canada");

 // Add language as a custom attribute
canada.attrib["Language"] = "French";

// Add the date of British North America Act, 1867
canada.attrib["ConstitutionDate"] = new DateTime(1867, 7, 1);

// Add the land area in km2
canada.attrib["AreaKm2"] = 9984670;
```

As you can see, a new property called "attrib" has been added to each country (same for provinces and cities). The attrib property is in fact a JSONObject capable or parsing and printing jSON-compliant data. It supports basic types like numbers and string, also dates and booleans, arrays and other JSON objects.

The attrib property is indexed so you can access the top-level fields of the JSONObject by its name or index number. To retrieve the values of the custom attributes added above, you'd do:

```
string language = canada.attrib["Language"]);
DateTime constitutionDate = canada.attrib["ConstitutionDate"].d; // Note the use of .d to force cast the internal number representation to DateTime
float countryArea = canada.attrib["AreaKm2"]);
```

Filtering by Custom Attributes

You can also launch a filtered seach of countries that match a predicate using Custom Attributes:

```
List<Country> countries = map.GetCountries(
        (attrib) => "French".Equals(attrib["Language"]) && attrib["AreaKm2"] > 1000000
        );
Int matchesFound = countries.Count);
```

The expression in bold is the predicate, expressed in lambda syntax. The GetCountries method as been overloaded so when you pass a lambda expression as above, it will iterate through all countries and pass its attrib JSONObject to your code, so you can interrogate it and return true if it matches your condition or not. In the example above, the lambda expression tests if the attributes passed contains a field Language which equals to "French" and also checks if the attribute "AreaKm2" is greater than 1000000.
Please note that you should check if the attributes list contains the field otherwise it will produce null exceptions in your predicate.

Importing / Exporting to JSON

The attrib object can parse a JSON-formatted string:

canada.attrib = new JSONObject(json); *// Import from raw jSON string*
int keyCount = canada.attrib.keys.Count; // Get the number of fields

And you can export current attributes of one country back to a JSON-formatted string:

string json = canada.attrib.Print();

To get the JSON-formatted string including all countries, you can call:

string jsonCountries = map.GetCountriesAttributes (true); // the true parameter will make the JSON string "pretty" (ie. adds tabs and end-of-line characters).

To get the JSON-formatted string including all countries, you can call:

string jsonCountries = map.GetCountriesAttributes (true); // the true parameter will make the JSON string "pretty" (ie. adds tabs and end-of-line characters).

To read all countries attributes from a custom string variable, call SetCountriesAttributes:

map.SetCountriesAttributes(jsonCountries);

The above method is called when you set the **countryAttributeFile** property.

Managing Custom Attributes

Custom Attributes added or modified will be lost if not persisted to file.

Using the Map Editor, you can manage custom attributes to countries, provinces, cities and mount points and save them to the Geodata folder (click "Save" button in the Map after making any change). *Beware that running your application without Saving changes will result in losing your changes!*

If you want to make changes to attributes and save them, you can get the JSON-formatted string as seen above for all attributes of all countries, provinces and ciites, and store it in your own database, file system, as user prefs, …

You can also have different custom attributes files. To reload the attributes file, just set the property countryAttributesFile (or provinceAttributesFile, cityAttributesFile) to a different name. The asset will try to find and load the data in the new file*. This file needs to be located inside Geodata folder.*

Loading and Saving data (demo scene 104)

Suppose your game changes geodata at runtime. It could change country name, or population, or the relationship between a province and its country, or even change frontiers of countries and provinces. Then you want to save the modified data so the player can resume the game in the future, how to do that?

Well, WMSK does not provide you a persistence layer, like a local database, but it provides you the required functions so you can get and set the state of the entities (countries, provinces, cities and mount points) in a formatted string so you can store that text information wherever you want and reload it in a later moment.

The functions are:

Countries

GetCountryGeoData(): returns a string containing the countries information, including the frontiers data.
SetCountryGeoData(string): loads the countries information, including frontiers data, from the given string.

Optional functions (only useful if you are adding custom attributes to countries):
GetCountriesAttributes(): returns a string in jSON format containing any country attribute (the contents of the .attrib field for each country).
SetCountriesAttributes(string): loads the countries attributes from the given string.

Provinces, Cities and MountPoints

Similar functions are available for provinces, cities and mountpoints (GetProvincesGeoData(), …).

Where can I store the geodata?

GetCountryGeoData(), GetProvinceGeoData(), … all returns a string with the required information. You can store this string in:

- A local text file in the Application data folder. Example: System.IO.
- A local database like SQLite.
- A remote database using a webservice.

For example, to save the country information to a local text file you could do:

```
string s = map.GetCountryGeoData();
System.IO.File.WriteAllText(Application.persistentDataPath + "/myCountries", s);
```

And to retrieve the country geodata in a future session you would do:

```
string s = System.IO.File.ReadAllText(Application.persistentDataPath + "/myCountries");
```

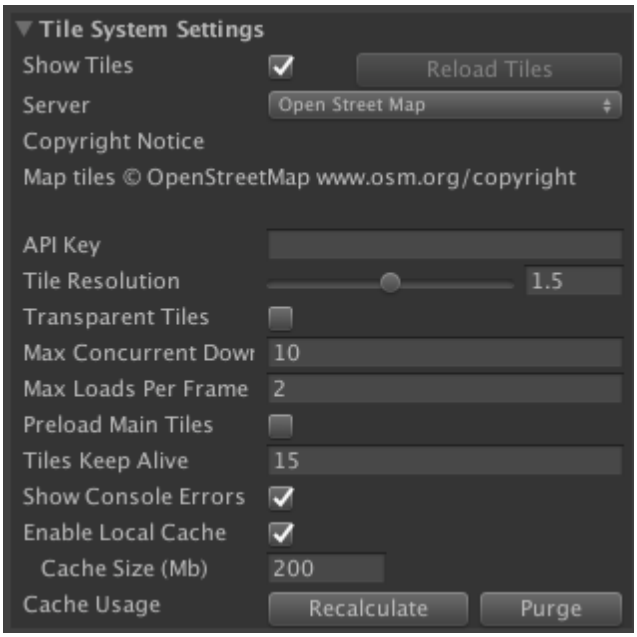
```
map.SetCountryGeoData(s);
```

Integrating with Online Tile Map Systems (demo scenes 06 and 514)

Since V7, World Map Strategy Kit can integrate with different tile servers to enable new Earth styles that provides progressive zoom detail. To enable this feature just enable the Tile system in the inspector.

Once you enable "Show Tiles" under Tile System section, the asset will automatically begin showing tiles based on current view and distance to Earth.

Options displayed in this section are:



Server: choose one of the provided tile servers. You can extend the list by editing the file WMSKTileServers.cs inside the Scripts folder.

Copyright notice: this is the copyright notice you should show in your applications according to the license you may have purchased from online map systems (see API Key below).

API Key: each server provides their tiles subject to terms of use. Most servers allow you to use them for free as long as you put their copyright notices visible in your application UI. Please refer to each server documentation online as their terms of use can change. Some servers may require you to sign up and obtain an API key or even purchase a special license if you exceed usage limitation for free tiers.

Currently the tile severs included are:

Tile Services	Terms of use / Copyright page
OpenStreeMap	http://www.osm.org/copyright
Stamen	http://maps.stamen.com
Carto	https://carto.com/location-data-services/basemaps/
Wikimedia Atlas	https://wikimediafoundation.org/wiki/Maps_Terms_of_Use
Thunderforest	http://thunderforest.com/terms/
OpenTopoMap	https://opentopomap.org/credits
MapBox	https://www.mapbox.com/pricing/
Sputnik	http://corp.sputnik.ru/maps
AerisWeather	https://aerisweather.com

Transparent Tiles: when enabled, all tiles will use a transparent material which enables seeing through them the map texture revealing the background map texture.

Tile Resolution: determines the maximum scaling when zooming in. A higher value will provide the sharpest resolution but also will lead to more tile downloads.

Max Concurrent Downloads: determines the maximum number of tile downloads at any given time. This value can be increased depending of the quality and bandwidth of your Internet connection.

Max Loads Per Frame: this is the maximum number of tiles showing up per frame. This option just refers to how many tiles will be activated per frame once they have been downloaded. When one tile is downloaded it's activated, and it appears on the map with a fade animation. This parameter controls the maximum number of animations started per frame.

Show Console Errors: if enabled, any tile request or download error will be printed out to the console if running inside Unity Editor or to the player.log file if running in a build).

Enable Local Cache: stores downloaded tiles locally into your device. The local cache size defaults to 50 Mb. The cache will automatically remove older tiles. You may increase this value to allow offline tile browsing.

Cache Usage: press Recalculate to display current usage of the local cache space. Press Purge to to remove those files.

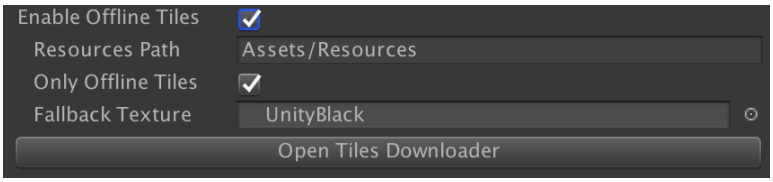
Note that you have also access to a tile API set of functions and properties which you can use to control and customize the tile system feature with scripting.

Downloading and embedding tiles with your application

It's possible to download and load map tiles from your application bundle directly. World Map Strategy Kit includes a Tile Downloader assistant that enables you to select a zoom level range and world area and fetch the tiles to a custom Resources folder inside your Unity application.

Each tile is a small PNG image file with name z_x_y.png referring to the zoom level and x/y tile coordinates. Tiles are stored in a subfolder with a number corresponding to the tile server enum. Please note that number of tiles grow exponentially with the zoom level. For a table of number of tiles per zoom level refer to https://wiki.openstreetmap.org/wiki/Tile_disk_usage

The new options are located in the Tile System section of WMSK's inspector:

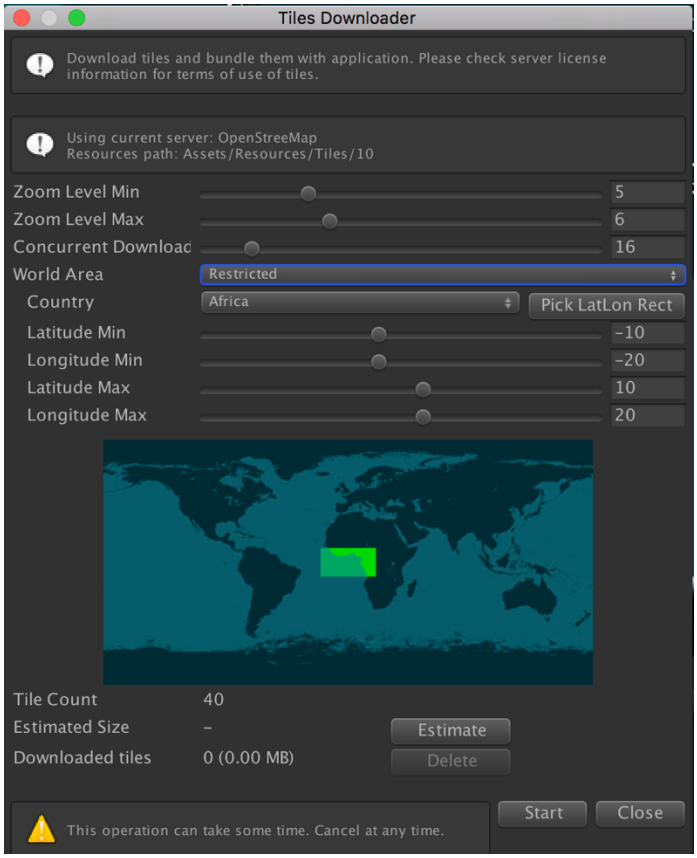


Resources Path: specify the root in the application project where the downloaded tiles will be stored. This path must be contained within the Assets folder at any level but must end with the "Resources" word.

Only Offline Tiles: if enabled only tiles available in the resource path will be used. If the user zooms or navigate to an area where no tiles are available, the **Fallback texture** will be used for those tiles.

Important: it's recommended to set the Max Zoom Level setting to the higher zoom level of the downloaded tiles.

Click "Open Tiles Downloader" to open the downloader assistant:



The downloader assistant shows the following options:

Zoom Level Min / Max: the range of zoom levels to download.

Concurrent Downloads: maximum number of simultaneous downloads. If your Internet connection and tile server can afford it, you can increase this number to download more tiles per second.

World Area: choose between “Full World” or “Restricted”. The restricted mode will download tiles within a given rectangle defined by latitude/longitude. For convenience a list of countries are shown which you can select to quickly select their rect area in the world.

Tile Count: this is the number of tiles that will be downloaded according to the range of zoom levels and selected world area.

Estimated Size: click “Estimate” to download a sample of tiles and produce an estimation of storage size for all the tiles. This is a very rough number and the actual size may vary.

Downloaded Tiles: total number and size of currently downloaded tiles.

Remarks:

- Use a limited zoom level to download tiles (ie. 5-6). As number of tiles grow exponentially with the zoom level, it’s not practical to include higher zoom levels unless you restrict the area.
- The downloader can be stopped at any time. When you click “Start” it will not download/replace any previously downloaded tile, so work can continue.

- If “Only Offline Tiles” is not checked, then the Tile System will try to load the tile from the Resources path. If not found, it will search in the Local Cache (if cache is enabled). Finally it will try to download it from the remote tile server.

Usage tips:

- If you just want to accelerate the load of the tiles in your application, you can download the tiles for only zoom level 5. Do NOT enable the “Only Offline Tiles” so the application will automatically download tiles for higher zoom levels.
- If you want to limit your application to a restricted set of tiles, then enable the “Only Offline Tiles” and make sure you download all the tiles using the Tiles Downloader. Remember that the Tiles Downloader assistant won’t remove any downloaded tile unless you click the “Delete” button. This behaviour allows you to combine different rectangle areas per different zoom levels.

Included Fonts

WMSK includes several free fonts inside Resources/Font folder. You may remove or add your own fonts as desired. When using a new font, make sure you set its font size to 160. To assign a new font to the map asset, just drag and drop it into the Font property in the inspector (or use the circle selector next to it).

Reducing game size

To reduce the size of your game, you can completely remove the following stuff:

- Demos folder.
- Textures of styles not used in Resources/Textures folder.
- Provinces data from Resources/Geodata folder if you don't use province borders.

API Reference Guide

You can instantiate the prefab "WorldMapStrategyKit" or add it to the scene from the Editor. Once the prefab is in the scene, you can access the functionality from code through the static instance property:

```
Using WorldMapStrategyKit;

WMSK map;

void Start () {
map = WMSK.instance;
        ...
}
```

(Note that you can have more WMSK instances in the same scene. In this case, the instance property will returns the same object. To use the API on a specific instance, you can get the WMSK component of the GameObject).

All public API and properties are located in WMSK_* scripts inside Scripts folder.

Map Entitites

Most of API methods and properties are related to one of the core map entities. These are core classes that store important information about the map or its contents.

Country class

The country class describes one country and its land regions. You can get an array of countries using **map.countries** property and/or using any method like **GetCountryIndex**…. Each country object in this array has the following properties and methods:

- **country.name**: the name of the country. This is used as a key internally so don't change it. If you want to display a different label, use customLabel property.
- **country.hidden**: if the country and its frontiers is visible in the map.
- **country.regions**: list of land regions of the country. A country is no more than the administrative info. The physical definition of its regions are inside this property.
- **country.mainRegionindex**: the index of the biggest region in the regions list.
- **country.regionsRect2D**: rect2D that encloses all country regions in the map. Used internally for country mouse detection.
- **country.center**: the center of the biggest region of the country in local space coordinates.
- **country.centerRect**: the geometric center of the rectangle enclosing all country regions in local space coordinates.

- **country.continent**: name of the continent to which the country belongs.
- **country.capitalCityIndex**: index of the capital city or -1 if none exists.
- **country.fips10_4, iso_a2, iso_a3, iso_n3:** standardized codes of the country.
- **country.provinces**: list of provinces of the country.
- **country.customLabel**: alternate label for the country. By default, the map will display the name but you can assign a different caption to this property.
- **country.labelColorOverride**: set this to true to use a custom color for this country label.
- **country.labelColor**: the custom color for this country label.
- **country.labelFontOverride**: specifies a different font for this country label.
- **country.labelVisible**: toggles visibility of this country label.
- **country.labelRotation**: custom rotation for this country label.
- **country.labelOffset**: custom offset for this country label.
- **country.labelFontSizeOverride**: set this to true to use a custom font size.
- **country.labelFontSize**: the custom font size for this country label.
- **country.uniqueId**: unique identifier of the country. Can be used as a key to relate this country with your custom classes.
- **country.attrib**: user defined attributes. This is a jSON object.
- **country.canCross**: determines if path finding methods can cross this country (only used with FindRoute when passing country objects).
- **country.crossCost**: used by path finding methods as an extra cost to pass this country (only used with FindRoute when passing country objects).
- **country.allowHighlight**: if this country can be highlighted.
- **country.showProvinces**: if provinces for this country will be drawn. Defaults to true.
- **country.allowProvincesHighlight**: if provinces for this country can be highlighted. Defaults to true.

Province class

The province class describes one province and its land regions. You can get an array of provinces using **map.provinces** property and/or using any method like **GetProvinceIndex**…. Each province object has the following properties and methods:

- **province.name**: the name of the province. This is used as a key internally so don't change it.
- **province.regions**: list of land regions of the province. A province is no more than the administrative info. The physical definition of its regions are inside this property.
- **province.mainRegionindex**: the index of the biggest region in the regions list.
- **province.regionsRect2D**: rect2D that encloses all province regions in the map. Used internally for province mouse detection.
- **province.center**: the center of the biggest region of the province in local space coordinates.
- **province.centerRect**: the geometric center of the rectangle enclosing all province regions in local space coordinates.
- **province.countryIndex**: the index of the country to which the province belongs.
- **province.uniqueId**: unique identifier of the province. Can be used as a key to relate this province with your custom classes.
- **province.attrib**: user defined attributes. This is a jSON object.
- **province.canCross**: determines if path finding methods can cross this province (only used with FindRoute when passing country objects).
- **province.crossCost**: used by path finding methods as an extra cost to pass this province (only used with FindRoute when passing province objects).
- **province.allowHighlight**: if this province can be highlighted.

City class

The city class describes one city. You can get an array of cities using **map.cities** property and/or using any method like **GetCityIndex**…. Each city object has the following properties and methods:

- **city.name**: the name of the city.
- **city.province**: name of the province where the city is located (optional).
- **city.countryIndex**: the index of the country to which the city belongs.
- **city.unity2DLocation**: location of the city in the map in local space coordinates (-0.5 .. 0.5).
- **city.population**: metropolitan population in thousands.
- **city.cityClass**: class of city (normal, region capital or country capital).
- **city.isVisible**: if the city is currently visible (drawn) in the map.
- **city.uniqueId**: unique identifier of the city. Can be used as a key to relate this city with your custom classes.
- **city.attrib**: root for custom attributes.

Mount Point class

The mount point class describes one user-defined location in the map. Note that mount points are not visible at such in the map – it's up to you to add any custom sprite or game object to their locations if you wish. You can get an array of mount points using **map.mountPoints** property and/or using any method like **GetMountPointIndex**…. Each mount point object has the following properties and methods:

- **mountpoint.name**: the name of the mountpoint. User-defined.
- **mountpoint.type**: type of mount point. User-defined.
- **mountpoint.countryIndex**: the index of the country to which the mount point belongs.
- **mountpoint.provinceIndex**: the index of the province to which the mount point belongs.
- **mountpoint.unity2DLocation**: location of the mount point in the map in local space coordinates (-0.5 .. 0.5).
- **mountpoint.uniqueId**: unique identifier of the mountpoint. Can be used as a key to relate this mountpoint with your custom classes.
- **mountpoint.attrib**: root for custom attributes.

Cell class

The cell class describes one cell of the hexagonal grid (when enabled). You can get an array of cells using **map.cells**. Each cell object has the following properties and methods:

- **cell.row / cell.column**: the location of the cell in the grid.
- **cell.center**: location of the cell in the map in local space coordinates (-0.5 .. 0.5).
- **cell.attrib**: root for custom attributes.

Events

General map events

public event OnMouseClick **OnClick**;
public event OnMouseClick **OnMouseDown;**
public event OnMouseClick **OnMouseRelease**;
public event OnMouseEvent **OnMouseMove**;
public event OnSimpleMapEvent **OnFlyTo**;
public event OnSimpleMapEvent **OnFlyEnd**;

Country events

public event OnCountryEvent **OnCountryEnter**;
public event OnCountryEvent **OnCountryExit**;
public event OnCountryClick **OnCountryClick**;
public event OnCountryHighlight **OnCountryHighlight**;

Province events

public event OnProvinceEvent **OnProvinceEnter**;
public event OnProvinceEvent **OnProvinceExit**;
public event OnProvinceClick **OnProvinceClick**;
public event OnProvinceHighlight **OnProvinceHighlight**;

Region (country or province) events

public event OnRegionClickEvent **OnRegionClick;**
public event OnRegionEnterEvent **OnRegionEnter;**
public event OnRegionEnterEvent **OnRegionExit;**

City events

public event OnCityEnter **OnCityEnter**;
public event OnCityEnter **OnCityExit**;
public event OnCityClick **OnCityClick**;

Grid cell events

public event OnCellEnter **OnCellEnter**;
public event OnCellExit **OnCellExit**;
public event OnCellClick **OnCellClick**;

Path-Finding events

public event OnPathFindingCrossPosition **OnPathFindingCrossPosition**;
public event OnPathFindingCrossAdminEntity **OnPathFindingCrossCountry;**
public event OnPathFindingCrossAdminEntity **OnPathFindingCrossProvince;**
public event OnPathFindingCrossAdminEntity **OnPathFindingCrossCell**;

Events associated with gameobjects added to the map

public Action<GameObjectAnimator> **OnVGOPointerDown**;
public Action<GameObjectAnimator> **OnVGOPointerUp**;
public Action<GameObjectAnimator> **OnVGOPointerEnter**;
public Action<GameObjectAnimator> **OnVGOPointerDown**;
public Action<GameObjectAnimator> **OnVGOMoveStart**;
public Action<GameObjectAnimator> **OnVGOMove**;
public Action<GameObjectAnimator> **OnVGOMoveEnd**;
public Action<GameObjectAnimator> **OnVGOCountryEnter**;
public Action<GameObjectAnimator> **OnVGOCountryRegionEnter**;
public Action<GameObjectAnimator> **OnVGOProvinceEnter**;
public Action<GameObjectAnimator> **OnVGOProvinceRegionEnter**;
public Action<GameObjectAnimator> **OnVGOKilled**;

Events associated with sprites added to the map

Get the MarkerClickHandler component from the sprite (check demo scene 501 Viewport Intro for a working example):

```
map.AddMarker2DSprite(star, planeLocation, 0.02f, enableEvents:true);
MarkerClickHandler handler = star.GetComponent<MarkerClickHandler>();
handler.OnMarkerMouseDown += (buttonIndex => Debug.Log("Click on sprite with button " +
buttonIndex + "!"));
```

public OnMarkerPointerClickEvent **OnMarkerMouseDown**;
public OnMarkerPointerClickEvent **OnMarkerMouseUp**;
public OnMarkerEvent **OnMarkerMouseEnter**;
public OnMarkerEvent **OnMarkerMouseExit**;
public OnMarkerEvent **OnMarkerDragStart;**
public OnMarkerEvent **OnMarkerDragEnd;**

Public Properties & Methods

General functions

map.ClearAll(): destroys all map data including frontiers, provinces, cities, mountpoints and any drawn surface. Useful to initialize and create a map procedurally using scripting.

map.paused: pauses the game ON/OFF.

map.timeSpeed: controls the speed of time flow. Defaults to 1.

map.cacheMaterials: by default WMSK caches all materials generated by coloring functions, so for instance, when using the same color to cover another cell, WMSK will reuse the same material. If you want to disable this behaviour and force WMSK to create a new material per cell, country or province regardless of the color, set this property to false.

Country related

map.countries: the array of Country objects. Note that the number and indexes of countries varies between the low and high-definition geodata files (reloaded when you change the frontiersQuality property in the inspector or in the API).

map.GetCountryIndex(name): returns the index of the country in the array.

map.GetCountry(index): returns the country object by its index. Equals to map.countries[index].

map.GetCountry(name): returns the country object by its name or null if not found.

map.GetCountryIndex(ray, out countryIndex, out regionIndex): find the country pointed by the ray.

map.GetCountryIndex(Vector2 localPosition): returns de country index that contains given local position.

map.GetCountryIndexByFIPS10_4(fips): returns the index of the country in the collection by FIPS code.

map.GetCountryIndexByISO_A2(iso_a2): returns the index of the country in the collection by ISO 2-character code.

map.GetCountryIndexByISO_A3(iso_a3): returns the index of the country in the collection by ISO 3-character code.

map.GetCountryIndexByISO_N3(iso_n3): returns the index of the country in the collection by ISO 3-digit code.

map.GetCountryCenter(int countryIndex): returns the geographic center of all country regions.

map.GetCountryNames(groupByContinent): returns an array with the country names, optionally grouped by continent.

map.GetVisibleCountries (): returns a list of countries that are completely or partly visible on the screen.

map.GetVisibleCountriesInWindowRect (): returns a list of countries that are inside the window rectangle (optional rect constraint set in the inspector or using windowRect).

map.countryHighlighted: returns the Country object for the country under the mouse cursor (or null).

map.countryHighlightedIndex: returns the index of country under the mouse cursor (or null if none).

map.countryRegionHighlighted: returns the Region object for the highlighted country (or null). Note that many contries have more than one region. The field mainRegionIndex of the Country object specified which region is bigger (usually the main body, being the rest islands or foreign regions).

map.countryRegionHIghlightedIndex: returns the index of the region of currently highlighted country for the regions field of country object.

map.countryLastClickedIndex: returns the index of last country clicked.

map.countryRegionHighlightedShape: returns the shape (gameobject) of the currently highlighted region.

map.enableCountryHighlight: set it to true to allow countries to be highlighted when mouse pass over them.

map.highlightAllCountryRegions: whether all regions of active country should be highlighted or just the one under the pointer (some countries can have more than one land region).

map.fillColor: color for the highlight of countries.

map.showCountryNames: enables/disables country labeling on the map.

map.showOutline: draws a border around countries highlightes or colored.

map.outlineColor: color of the outline.

map.showFrontiers: show/hide country frontiers. Same than inspector property.

map.frontiersDetail: detail level for frontiers. Specify the frontiers catalog to be used.

map.frontiersColor: color for all frontiers.

map.frontiersDynamicWidth: changes the width of the country frontiers automatically based on distance of camera to the map.

map.RenameCountry(oldName, newName): allows to change the country's name. Use this method instead of changing the name field of the country object.

map.BlinkCountry(country, color1, color2, duration, speed): makes the country specified toggle between color1 and color2 for duration in seconds and at speed rate.

map.FlyToCountry(name): start navigation at *navigationTime* speed to specified country. The list of country names can be obtained through the cities property.

map.FlyToCountry(index): same but specifying the country index in the countries list.

map.ToggleCountrySurface(name, visible, color): colorize one country with color provided or hide its surface (if visible = false).

map.ToggleCountrySurface(index, visible, color): same but passing the index of the country instead of the name.

map.ToggleCountryMainRegionSurface(index, visible, color, Texture2D texture): colorize and apply an optional texture to the main region of a country.

map.ToggleCountryRegionSurface(countryIndex, regionIndex, visible, color): same but only affects one single region of the country (not province/state but geographic region).

map.HideCountrySurface(countryIndex): un-colorize / hide specified country.

map.HideCountryRegionSurface(countryIndex): un-colorize / hide specified region of a country (not province/state but geographic region).

map.HideCountrySurfaces: un-colorize / hide all colorized countries (cancels ToggleCountrySurface).

map.ToggleCountryOutline(countryIndex, visible, texture, borderWidth, tintColor, …): draws an outline around the given country (or hide it if visible is set to false). This method includes all regions of the country.

map.ToggleCountryMainRegionOutline(countryIndex, visible, texture, borderWidth, …): draws an outline around the given country (or hide it if visible is set to false). Only affects the main region of the country.

map.ToggleCountryRegionOutline(countryIndex, regionIndex, visible, texture, borderWidth, …): draws an outline around the given country (or hide it if visible is set to false). Only affects specified region index.

map.CountryNeighbours(countryIndex): returns the list of country neighbours.

map.CountryNeighboursOfMainRegion(countryIndex): same but taking into account only the main region of the province (just in case the province could have more than one land region).

map.CountryNeighboursOfCurrentRegion(): same but taking into account the currently highlighted province.

map.CountryCreate(name, continent): returns a new country object with name and continent (name must be unique!).

map.CountryAdd(country): adds a new country object created with "New Country(…)" instead of Country Create.

map.countryAttributeFile: name of the resource jSON file storing attributes for countries.

map.GetCountryGeoData(): returns a string with all countries frontiers and data.

map.SetCountryGeoData(): loads countries frontiers and data from a string.

map.GetCountriesAttributes(prettyPrint): returns a JSON-formatted string with all attributes for all countries.

map.GetCountriesAttributes(countries, prettyPrint): same but for a list of countries.

map.SetCountriesAttributes(jSON): sets the attributes of a list of countries contained in the JSON-formatted string.

map.GetCountryCoastalPoints(countryIndex): returns a list of map position where the coast of a country is.

map.GetCountryFrontierPoints(countryIndex1, regionIndex, worldSpace): returns a list of points of the given country region. Optionally in world space.

map.GetCountryFrontierPoints(countryIndex1, countryindex2): returns a list of map position where two given countries share frontiers.

map.GetCountryZoomExtents(index): gets the required zoom level to show a custom country within screen borders (including all its regions! If some regions are far away, it might not zoom properly since entire or large part of the world must be visible!).

map.GetCountryRegionZoomExtents(index, region): gets the required zoom level to show a custom country within screen borders (this will only include the main region or a given region.

map.GetCountryRegionSurfaceGameObject(countryIndex, regionIndex): returns the cached surface of the colored country (if not colored, this function will return null).

map.GetCountryRegion(localPosition): returns the region object located at given map coordinates.

map.CountryTransferCountry (countryIndex, sourceCountryIndex, redraw): source country identified by sourceCountryIndex is conquered/absorbed by countryIndex. Redraw parameter optionally redraws frontiers to reflect the new country perimeters.

map.CountryTransferCountryRegion(countryIndex, sourceCountryRegion, redraw): same than before but using a source region object (it transfer entire country though). SourceCountryRegion is conquered/absorbed by countryIndex. Redraw parameter optionally redraws frontiers to reflect the new country perimeters.

map.CountryTransferProvinceRegion (countryIndex, provinceRegion, redraw): province is absorbed by the given countryIndex. Redraw parameter optionally redraws frontiers to reflect the new country perimeters.

map.CountryTransferCell (countryIndex, cellIndex, redraw): cell is absorbed by the given countryIndex. Redraw parameter optionally redraws frontiers to reflect the new country perimeters.

map.CountryRemoveCell (countryIndex, cellIndex, redraw): cell is removed from target country. Redraw parameter optionally redraws frontiers to reflect the new country perimeters.

map.CountryDeleteProvinces (countryIndex): remove all provinces from a given country.

Region related (common to countries and provinces)

map.GetRegionSurfaceGameObject(region): returns the cached surface of the colored region, either it belongs to a country or a province (if not colored, this function will return null).

map.GetRegionColor(region): returns the current color of a region or transparent if not colored (transparent = Color(0,0,0,0)).

map.GetRegionOverlap(region, includeProvinces): returns a list of overlapping regions with a given region optionally including provinces regions.

map.RegionErase (region, eraseColor): erases/paints the texture portion occupied by the region with a given color.

map.RegionSetCustomElevation (List<region>, elevation): sets a custom elevation for the given regions. Works only with viewport mode.

map.RegionRemoveCustomElevation (List<region>, elevation): clears any previous custom elevation for the given regions. Works only with viewport mode.

map.RegionRemoveAllCustomElevation (): removes any custom elevation assigned to any region. Works only with viewport mode.

map.RegionGenerateExtrudeGameObject (name, region, amount, …): creates a new extruded region and position it on top of original region.

Province related

map.provinces: return a List<Province> of all provinces/states records.

map.provinceHighlighted: returns the province/state object in the provinces list under the mouse cursor (or null if none).

map.provinceHighlightedIndex: returns the province/state index in the provinces list under the mouse cursor (or null if none).

map.provinceRegionHighlighted: returns the highlighted region of the province/state (or null).

map.provinceRegionHighlightedIndex: returns the index of the region highlighted for the regions field of province object).

map.provinceLastClicked: returns the last province clicked by the user.

map.provinceRegionLastClicked: returns the last province's region clicked by the user.

map.provinceRegionHighlightedShape: returns the shape (gameobject) of the currently highlighted region.

map.showProvinces: show/hide provinces when mouse enters a country. Same than inspector property.

map.provincesFillColor: color for the highlight of provinces.

map.enableProvinceHighlight: whether provinces will be highlighted when the pointer pass over them.

map.highlightAllProvinceRegions: whether all regions of active province should be highlighted or just the one under the pointer (some provinces can have more than one land region).

map.provincesColor: color for provinces/states color.

map.GetProvinceIndex(name): returns the index of the province in the array. Please note that there some provinces with same name. You may want to use GetProvinceIndex(country, name) instead.

map.GetProvinceIndex(country, name): returns the index of the province inside the province array of the country object.

map.GetProvinceIndex (Vector2 localPosition): returns de province index that contains given local position.

map.GetProvince(index): returns the province object by its index. Equals to map.provinces[index].

map.GetProvince(provinceName, countryName): returns the province object by its name for the country given or null if not found.

map.GetProvinceCenter(int provinceIndex): returns the geographic center of all province regions.

map.GetProvinceNames(groupByCountry): returns an array with the province names, optionally grouped by country.

map.GetVisibleProvinces (): returns a list of provinces that are visible on the screen.

map.GetVisibleProvincesInWindowRect (): returns a list of provinces that are inside the window rectangle (optional rect constraint set in the inspector or using windowRect).

map.RenameProvince(oldName, newName): allows to change the province's name. Use this method instead of changing the name field of the province object.

map.DrawProvinces(name, includeNeighbours, forceRefresh): draws borders for the province specified. Optionally can add the neighbouts province's borders. ForceRefresh is used internally – usually pass false.

map.HideProvinces(): hides the provinces borders.

map.BlinkProvince(province, color1, color2, duration, speed): makes the province specified toggle between color1 and color2 for duration in seconds and at speed rate.

map.FlyToProvince(name): start navigation at *navigationTime* speed to specified province. The list of provinces names can be obtained through the provinces property.

map.FlyToProvince (index): same but specifying the province index in the provinces list.

map.ToggleProvinceSurface(name, visible, color): colorize one province with color provided or hide its surface (if visible = false).

map.ToggleProvinceSurface(index, visible, color): same but passing the index of the country instead of the name.

map.HideProvinceSurface(countryIndex): un-colorize / hide specified province.

map.HideProvinceSurfaces: un-colorize / hide all colorized provinces (cancels ToggleProvinceSurface).

map.ProvinceCreate(name, countryIndex): created a new province for a given country. Name must be unique!

map.ProvinceAdd(province): adds a new province object created with "New Province(…)" instead of Country Create.

map.ProvinceRename(country, oldName, newName): renames an existing province.

map.ProvinceNeighbours(provinceIndex): returns the list of province neighbours.

map.ProvinceNeighboursOfMainRegion(provinceIndex): same but taking into account only the main region of the province (just in case the province could have more than one land region).

map.GetProvinceRegionSurfaceGameObject(provinceIndex, regionIndex): returns the cached surface of the colored province (if not colored, this function will return null).

map.ProvinceNeighboursOfCurrentRegion(): same but taking into account the currently highlighted province.

map.ProvinceToCountry(province, newCountryName): creates a new country based on an existing province. The province is extracted from the original country.

map.ProvinceTransferProvinceRegion (provinceIndex, provinceRegion, redraw): province region is absorbed by the given province. Redraw parameter optionally redraws frontiers to reflect the new country perimeters.

map.ProvinceTransferCell (provinceIndex, cellIndex, redraw): cell is absorbed by the given province. Redraw parameter optionally redraws frontiers to reflect the new province perimeters.

map.ProvinceRemoveCell (provinceIndex, cellIndex, redraw): cell is removed from target province. Redraw parameter optionally redraws frontiers to reflect the new province perimeter.

map.ProvinceDelete (provinceIndex): remove specified province.

map.provinceAttributeFile: name of the resource jSON file storing attributes for countries.

map.GetProvinceGeoData(): returns a string with all provinces frontiers and data.

map.SetProvinceGeoData(): loads provinces frontiers and data from a string.

map.GetProvincesAttributes(prettyPrint): returns a JSON-formatted string with all attributes for all provinces.

map.GetProvincesAttributes(countries, prettyPrint): same but for a list of provinces.

map.SetProvincesAttributes(jSON): sets the attributes of a list of provinces contained in the JSON-formatted string.

map.GetProvinceCoastalPoints(provinceIndex): returns a list of map position where the coast of a province is.

map.GetProvinceBorderPoints(provinceIndex, regionIndex, worldSpace): returns a list of points of the given province region. Optionally in world space.

map.GetProvinceBorderPoints(provinceIndex1, provinceindex2): returns a list of map position where two given provinces share borders.

map.GetProvinceZoomExtents(index): gets the required zoom level to show a custom province within screen borders (including all its regions! If some regions are far away, it might not zoom properly since entire or large part of the world must be visible!).

map.GetProvinceRegionZoomExtents(index): gets the required zoom level to show a custom province within screen borders.

map.GetRegionSurfaceGameObject(Region region): returns the cached surface of the colored region, either it belongs to a country or a province (if not colored, this function will return null).

map.GetRegionSurfaceColor(Region region): returns the current color of a region or transparent if not colored (transparent = Color(0,0,0,0)).

Cities related

map.cities: return a List<City> of all cities records.

map.GetCityNames: return an array with the names of the cities.

map.GetCityIndex(name): returns the city object by its name for the city name given or null if not found.

map.GetCity(cityName, countryName): returns the city object by its name for the country given or null if not found.

map.GetCityPosition(int provinceIndex): returns the map position of a city.

map.cityHighlighted: returns the city under the mouse cursor (or null if none).

map.cityHighlightedIndex: returns the city index under the mouse cursor (or -1 if none).

map.lastCityClicked: returns the city clicked by the user.

map.showCities: show/hide all cities. Same than inspector property.

map.minPopulation: the mínimum population amount for a city to appear on the map (in thousands). Set to zero to show all cities in the current catalog. Range: 0 .. 17000.

map.cityClassAlwaysShow: combiantion of bitwise flags to specify classes of cities that will be drawn irrespective of other filters like minimum population. See CITY_CLASS_FILTER* constants.

map.citiesColor: color for the normal cities icons.

map.citiesRegionCapitalColor: color for the region capital cities icons.

map.citiesCountryCapitalColor: color for the country capital cities icons.

map.FlyToCity(name): start navigation at *navigationTime* speed to specified city. The list of city names can be obtained through the cities property.

map.FlyToCity(index): same but specifying the city index in the cities list.

map.cityAttributeFile: name of the resource jSON file storing attributes for countries.

map.GetCityGeoData(): returns a string with all cities frontiers and data.

map.SetCityGeoData(): loads cities frontiers and data from a string.

map.GetCitiesAttributes(prettyPrint): returns a JSON-formatted string with all attributes for all cities.

map.GetCitiesAttributes(cities, prettyPrint): same but for a list of cities.

map.SetCitiesAttributes(jSON): sets the attributes of a list of cities contained in the JSON-formatted string.

map.GetCountryCapital(country): returns the capital of a given country.

map.CityAdd(City newCity): adds a new city to the map.

Earth related

map.showEarth: show/hide the planet Earth. Same than inspector property.

map.earthStyle: the currently texture used in the Earth.

map.earthColor: the current color used in the Earth when style = SolidColor.

map.earthTexture: the current texture used in the Earth when style = SolidTexture.

map.earthMaterial: returns current Earth material.

map.showLatitudeLines: draw latitude lines.

map.latitudeStepping: separation in degrees between each latitude line.

map.showLongitudeLines: draw longitude lines.

map.longitudeStepping: number of longitude lines.

map.gridLinesColor: color of latitude and longitude lines.

map.ToggleContinentSurface(name, visible, color): colorize countries belonging to specified continent with color provided or hide its surface (if visible = false).

map.HideContinentSurface(name): uncolorize/hide countries belonging to the specified continent.

map.waterColor: color of the water for the Scenic Plus style.

map.ContainsWater(position): returns true if specified position contains water (sea, river, lake, …)

map.ContainsWater(position, boxAreaSize, out waterPosition): same but applied to a box centered on position. Returns the position of the water found.

Viewport related

map.earthCloudLayer: enabled/disables the cloud layer when using viewport.

map.earthCloudLayerSpeed: cloud animation speed when cloud layer is enabled.

map.earthCloudLayerAlpha: cloud transparency when cloud layer is enabled.

map.earthCloudLayerShadowStrength: cloud shadow transparency when cloud layer is enabled.

map.fogOfWarLayer: enables/disables the fog of war layer.

map.fogOfWarColor: gets/sets the fog of war color.

map.FogOfWarClear(): resets the fog of war state and makes everything dark again.

map.FogOfWarGet(x,y): gets the transparency of the fog of war at a given map position.

map.FogOfWarSet(x,y,alpha): sets the transparency of the fog of war at a given map position.

map.FogOfWarIncrement(x,y,alphaIncrement, radius): changes (adds/substracts) by alphaIncrement the transparency of the fog of war inside a circle defined by a given map position and a radius.

map.FogOfWarSetCountry(countryIndex, alpha): sets the transparency of the fog of war over a given countr, including all of its regions.

map.FogOfWarSetCountryRegion(countryIndex, regionIndex,alpha): sets the transparency of the fog of war over a given country region.

map.FogOfWarSetProvince(provinceIndex, regionIndex,alpha): sets the transparency of the fog of war over a given province, including all of its regions.

map.FogOfWarSetProvinceRegion(provinceIndex, regionIndex,alpha): sets the transparency of the fog of war over a given province region.

map.fogOfWarTexture: gets/sets the fog of war texture. You can set a new, bigger texture to increase the resolution of the fog of war on the map.

map.sun: gets or sets the light gameobject that represents the sun for the day/night cycle.

map.timeOfDay: gets or sets the current time of day for the day/night cycle effect (0-24).

Map interaction and navigation

map.mouseIsOver: returns true if mouse has entered the Earth's sphere collider.

map.navigationTime: time in seconds to fly to the destination (see FlyTo methods).

map.allowUserDrag/map.allowUserZoom: enables/disables user interaction with the map.

map.SetZoomLevel(level): apply one-time zoom level from 0 (closest) to 1 (farther).

map.zoomMinDistance / map.zoomMaxDistance: limits the amount of zoom user can perform.

map.invertZoomDirection: controls the zoom in/out direction when using mouse wheel.

map.allowUserKeys/map.dragFlipDirection: enables/disables user drag with WASD keys and direction.

map.allowScrollOnScreenEdges: enables/disables autodisplacement of the map when mouse is positioned on the edges of the screen.

map.mouseWheelSensibility: multiplying factor for the zoom in/out functionality.

map.mouseDragSensibility: multiplying factor for the drag functionality.

map.showCursor: enables the cursor over the map.

map.cursorFollowMouse: makes the cursor follow the map.

map.cursorLocation: current location of graphical cursor in local coordinates (by default the sphere is size (1,1,1) so x/y/z can be in (-0.5,0.5) interval. Can be set and the cursor will move to that coordinate.

map.GetCurrentMapLocation:
returns the coordinates of the center of the map as it's shown on the screen.

map.fitWindowWidth: makes the map occupy the width of the screen.

map.fitWindowHeight: makes the map occupy also the height of the screen.

map.CenterMap(): positions the map in front of the main camera.

map.windowRect: current rectangle constraints. By default, the entire map is shown with windowRect being a rect of (-0,5, -0,5, 1, 1).

map.FlyToLocation (x, y, z): same but specifying the location in local Unity spherical coordinates.

map.respectOtherUI: if set to true, it will prevent interaction with the map while pointer is over another UI element.

map.cursorAlwaysVisible: set this to false to hide the cursor automatically when pointer is not over the map.

Labels related

map.showCountryNames: toggles countriy labels on/off.

map.countryLabelsSize: this is the relative size for labels. Controls how much the label can grow to fit the country area.

map.countryLabelsAbsoluteMinimumSize: minimum absolute size for all labels.

map.labelsQuality: specify the quality of the label rendering (Low, Medium, High).

map.showLabelsShadow: toggles label shadowing on/off.

map.countryLabelsColor: color for the country labels. Supports alpha.

map.countryLabelsShadowColor: color for the shadow of country labels. Also supports alpha.

map.countryLabelsFont: Font for the country labels.

Mount Points related

map.mountPoints: return a List<MountPoint> of all mount points records.

map.GetMountPointNearPoint: returns the nearest mount point to a location on the sphere.

map.GetMountPoints: returns a list of mount points, optionally filtered by country and province.

Markers & Lines

map.AddMarker2DSprite(marker, planeLocation, markerScale, enableEvents, autoScale): adds a sprite (provided by you) to the map at plane location with specified scale (markerScale). The parameter enableEvents specifies if mouse events should be handled automatically (click events, dragging). The parameter autoScale specifies if the size of the sprite should be automatically adjusted based on the zoom level.

map.AddMarker3DSObject(marker, planeLocation, markerScale): adds a 3D gameobject (provided by you) to the map at plane location with specified scale (markerScale).

map.AddMarker2DText(text, planeLocation): adds a text to the map.

map.GetMarkers(List<Transform> results): returns all markers added to the map. This function is overloaded and you can optionally pass the Country, Province, Region or Cell object for which you want the markers. You must pass an initialized List and this function will fill in the results.

map.AddLine(start, end, color, elevation, drawingDuration, lineWidth, fadeAfter): adds an animated line to the map from start to end position and with color, elevation and other options. Check the AddLine section below for available options for lines.

map.AddCircle(position, kmRadius, ringWidthStart, ringWidthEnd, color, overdraw, renderOrder): adds a circle to the markers overlay at map position, radius, ring size (0..1) and color (alpha supported). Set overdraw to false to avoid colors of different overlapping circles to mix. The renderOrder can be used to control which circle is rendered before others

Hexagonal Grid

map.cells: linearized array of (gridRows * gridColumns) dimensions containing cells

map.showGrid: toggles on/off hexagonal grid.

map.gridRows / map.gridColumns: dimensions of the grid.

map.gridColor: color for the grid. It accepts transparency.

map.gridMinDistance / gridMaxDistance: distances in world units from the camera to the map where the grid is visibile. Outside of this range, the grid will fade out gracefully.

map.enableCellHighlight: toggles on/off cell highlighting.

map.cellHighlightColor: color for the highlighting effect.

map.cellHighlighted: returns the Cell object currently highlighted.

map.cellHighlightedIndex: returns the index of the currently highlighted cell.

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

map.GetCellindex(cell): returns the index of a given cell object.

map.ToggleCellSurface(cellIndex, visible, color, refreshGeometry): colorizes a cell by index according to visible and color parameters. Use refreshGeometry = true to ignore cached surfaces (if not sure, pass “false”).

map.ToggleCellSurface(cellIndex, visible, color, refreshGeometry, texture, textureScale, textureOffset, textureRotation): add a texture to a cell by index according to visible, color and texture parameters. Use refreshGeometry = true to ignore cached surfaces (if not sure, pass “false”).

map.HideCellSurface(cellIndex): programatically hides a colorized / textured cell.

map.CellFadeOut(cellIndex, color, duration): initiates a fading out effect on speficied cell with color and duration in seconds. When finished, the cell is hidden.

map.CellFadeOut(cells, color, duration): initiates a fading out effect on a group of cells with color and duration in seconds. When finished, the cell is hidden.

map.CellBlink(cellIndex, color, duration): initiates a blink effect on speficied cell with color and duration in seconds. The previous color is preserved.

map.CellBlink(cells, color, duration): initiates a blink effect on a group of cells with color and duration in seconds. The previous color is preserved.

map.CellFlash(cellIndex, color, duration): initiates a flash effect on speficied cell with color and duration in seconds. The previous color is preserved.

map.CellFlash(cells, color, duration): initiates a flash effect on a group of cells with color and duration in seconds. The previous color is preserved.

map.GetCellWorldPosition(cellIndex): returns center of the cell in world space coordinates (cell.center contains the local space coordinates).

map.GetCellVertexWorldPosition(cellIndex, vertexIndex): returns position of the specified vertex (0-5) of the cell in world space coordinates.

map.GetCell(localPosition): returns the cell which contains provided local coordiantes.

map.GetCellsInCountry(countryIndex): returns the cells belonging to a given country.

map.GetCellsInProvince(provinceIndex): returns the cells belonging to a given province.

map.GetCellCountry(cellIndex): returns the country index to which a cell belongs.

map.GetCellProvince(cellIndex): returns the province index to which a cell belongs.

map.GetCellNeighbours(cellIndex, …): returns a list of indices of cells that are neighbours or a given cell.

map.GetCellPathCost(cellIndex): returns the cost for crossing this cell computed in the last FindRoute function call.

map.GetCellsByAltitude(minAltitude, maxAltitude): returns a list of cells between min and max altitudes.

map.GetCellsWithinRadius(int cellIndex, int radius, List<int> cellIndices): returns the indices of cells found in a radius around cellIndex.

map.GetCellsWithinRectangle(int rowMin, int rowMax, int columnMin, int columnMax, List<int> cellIndices): returns the indices of cells found inside a given rectangle by row/column numbers.

map.GetCellsWithinCone(int cellIndex, Vector2 direction, float maxDistance, float angle, List<int>cellindices): returns the indices of cells found inside a cone starting at cellIndex in a given direction with a maximum distance.

map.GetCellsWithinDistanceKM(int cellIndex, float km, List<int> cellIndices): returns the indices of cells within km distance to another cell.

Path Finding related

map.pathFindingHeuristicFormula: formula for estimating cost from a given position to destination.

map.pathFindingMaxCost: maximum accumulated cost for finding routes. You can increase this value to return longer routes at performance expense. Default max value is 2000 cost points.

map.pathFindingMaxSteps: maximum number of steps for any route You can increase this value to return longer routes at performance expense. Default max value is 2000 steps.

Open World (non-grid based) movement pathfinding functions

map.pathFindingEnableCustomRouteMatrix: set this property to true to enable custom location costs. This will allow the usage of functions below to modify the custom route matrix. Basically the values in this matrix are added to the calculated crossing cost of the path finding engine for that location (a 0 will not add any cost).

map.PathFindingCustomRouteMatrixReset(): call this function to clear/reset current custom route matrix. Usually you call this function before populating the route matrix with your own custom values.

map.PathFindingCustomRouteMatrix (get/set): returns a copy of the current custom route matrix (or set it with a provided matrix). This is useful to store different cost matrix for each player and restore it at the start of each player's turn.

map.PathFindingCustomRouteMatrixSet (position, cost): specifies movement cost for a given map position.

map.PathFindingCustomRouteMatrixSet (List<Vector2> positions, cost): specifies movement cost for a list of map position. A cost of 0 means unbreakable position.

map.PathFindingCustomRouteMatrixSet (region, cost): specifies movement cost for a region (eg. a province or country region). A cost of 0 means unbreakable position.

map.PathFindingCustomRouteMatrixSet (province, cost): specifies movement cost for a province. A cost of 0 means unbreakable position.

map.PathFindingCustomRouteMatrixSet (country, cost): specifies movement cost for a country. A cost of 0 means unbreakable position.

map.FindRoute(startPosition, endPosition, terrainCapability, minAltitude, maxAltitude): returns an optimal route from startPosition to endPosition as a list of map positions (Vector2) having into account terrain capability, minimum altitude and maximum altitude options.

map.PathFindingGetProvincesInPath(path): returns a list of provinces indices being crossed by a path.

map.PathFindingGetCountriesInPath(path): returns a list of countries indices being crossed by a path.

map.PathFindingGetCitiesInPath(path): returns a list of cities indices being crossed by a path.

map.PathFindingGetMountPointsInPath(path): returns a list of mount points indices being crossed by a path.

Country to Country pathfinding functions

map.FindRoute(startCountry, destinationCountry, …): returns a list of Country objects that connect startCountry with destinationCountry including them.

Province to Province pathfinding functions

map.FindRoute(startProvince, destinationProvince, …): returns a list of Country objects that connect startCountry with destinationCountry including them.

Hexagonal Grid pathfinding functions

map.FindRoute(startCell, destinationCell, …): returns a list of cell indices that connect two cells.

map.PathFindingCellSetSideCost(cell, side, cost): sets a crossing cost for a cell's side.

map.PathFindingCellSetAllSidesCost(cell, cost): sets a crossing cost for all cell's sides.

map.PathFindingCellSetAllSidesCost(cell1, cell2, cost): sets a crossing cost between cell1 and cell2.

map.PathFindingGetSideCost(cell1, side, cost): gets the crossing cost for that cell's side.

map.PathFindingCustomCellCosts (get/set): returns a copy of the current custom cell costs (or set it with a provided array). This is useful to store different cost configuration for the entire grid for each player and restore it at the start of each player's turn or prior an unit movement which uses ifferent values.

LineMarkerAnimator component

The LineMarkerAnimator is responsible for drawing and animating lines generated by AddLine method. The AddLine method returns a reference to the LineMarkerAnimator which exposes many properties to customize the line appearance. Note that some of these properties are already set when you call AddLine.

color: the color of the line.
lineWidth: width of the line (defaults to 0.01f
arcElevation: elevation of the arc. A value of zero means the line will be drawn flat, on the ground.
drawingDuration: the duration for the drawing of the line. A value of 0 means instant drawing.
reverseMode: if set to true, line will start fully drawn and reduced over time.
lineMaterial: the material used to render the line. If not specified, the asset will use the default line marker material.
autoFadeAfter: the duration in seconds for the line before it fades out.
fadeOutDuration: the duration for the fade out effect until the line disappears.
dashInterval: the gap separation between dashes. A value of 0 means continuous line.
dashAnimationDuration: duration of a cycle in seconds. 0.1f can be a good value, 0 = no animation.
numPoints: the number of points used to draw the line. By default it uses 64 points.
startCap: the gameObject used for the start cap. Can be a model or sprite.
startCapFlipDirection: a Boolean that specifies the direction of the starting cap.
startCapOffset: a displacement for the starting cap.
startCapMaterial: material used for the staring cap. If omitted, it uses a default line marker material.
endcap / **endCalFlipDirection** / **endCapOffset** / **endCapMaterial**: same but for the opposite edge of the line.

Extra components accesors

map.calc: accesor to the Calculator component.

map.ticker: accesor to the Tickers component.

map.decorator: accesor to the Decorator component.

Geodata file format description

Data for countries, provinces and cities are stored in text files located in WorldMapPoliticalMapGlobeEdition/Resources/Geodata folder.

- countries10.txt: data for country frontiers in high resolution
- countries110.txt: data for country frontiers in low resolution
- provinces10.txt: data for province borders (always in high resolution)
- cities10.txt: data for cities.

All data is packed in string fields using separators. JSON, although more human-friendly format, is not used to reduce the file size and optimize the loading time (especially for mobile devices).

Country file format

1. Contains a list of countries separated by |.
   Example: COUNTRY0|COUNTRY1|COUNTRY2…)

2. Each COUNTRY entry is a series of fields separated by $:
   COUNTRY = {NAME $ CONTINENT $ REGIONS $ HIDDEN $ UNIQUE_ID $ FIPS 10.4 CODE $ ISO A2 $ ISO A3 $ ISO N3 }

   Name = name of the country (as displayed on the map)
   Continent = continent name (useful to group countries)
   Hidden = 0 (visible) or 1 (invisible)
   Unique_ID = unique id generated for this country
   FIPS / ISO codes = standard id codes for country
   Label visibility = 1 (visible) or 0 (invisible)

3. REGIONS is a list of polygons, separated by *:
   REGIONS = REGION0*REGION1*REGION2…

4. Each REGION is a list of latitude/longitude coordinates in the format:
   REGION = coord1; coord2 ; coord3 ; coord4 …

5. The coordinates format is x, y position in the range (-0.5, 0.5) divided multiplied by 5.000.000 and rounded, where -0.5 Is left or bottom edge of the map and 0.5 is top or right edge of map.

The method SetCountryGeoData() method in WPMCountries.cs is responsible of loading and extracting the file data.

Province file format

1. Contains a list of provinces separated by |.
   Example: PROVINCE0|PROVINCE1|PROVINCE2…)

2. Each PROVINCE entry is a series of fields separated by $:
   PROVINCE = {NAME $ COUNTRY_NAME $ REGIONS $ UNIQUE_ID }

   Name = name of the country (as displayed on the map)
   Country_Name = the name of the country to which the province belongs to
   Unique_ID = unique id generated for this country

3. REGIONS is a list of polygons, separated by *:
   REGIONS = REGION0*REGION1*REGION2…

4. Each REGION is a list of latitude/longitude coordinates in the format:
   REGION = coord1; coord2 ; coord3 ; coord4 …

5. The coordinates format is x, y position in the range (-0.5, 0.5) divided multiplied by 5.000.000 and rounded, where -0.5 Is left or bottom edge of the map and 0.5 is top or right edge of map.

The method SetProvincesGeoData() and ReadProvincePackedString() methods in WPMProvinces.cs are responsible of loading and extracting the file data.

City file format

1. Contains a list of cities separated by |.
   Example: CITY0|CITY1|CITY2…)

2. Each CITY entry is a series of fields separated by $:
   CITY = {NAME $ PROVINCE_NAME $ COUNTRY_NAME $ POPULATION $ X $ Y $ CLASS $ UNIQUE_ID }

   Name = name of the city
   Province_Name = the name of the province
   Country_Name = the name of the country to which the province belongs to
   X, Y = coordinate of the city (see country / province section above)
   Population = metropolitan population
   Class = 1 (regular city), 2 (region capital) or 4 (country capital)

The method SetCityGeoData() method in WPMCities.cs is responsible of loading and extracting the file data.

Additional Components

World Map Calculator

This component is useful to:

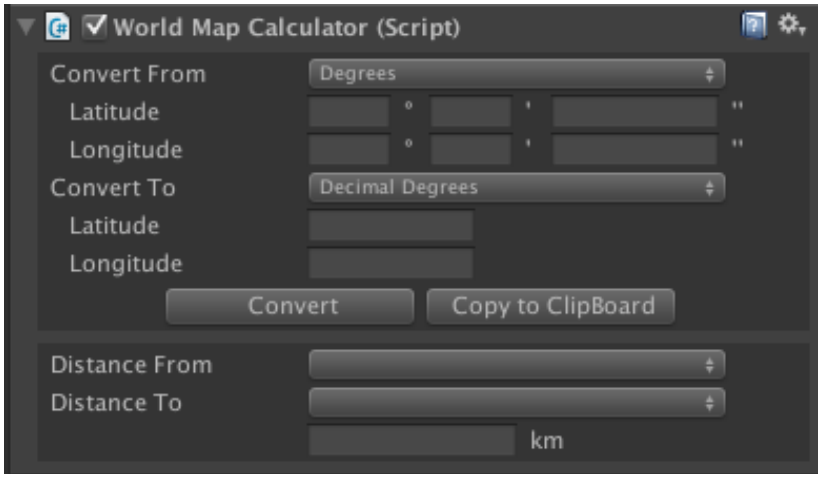
- Convert units from one coordinate system to another (for instance from plane coordinates to degrees and viceversa).
- Calculate the distance between cities.

You may also use this component to capture the current cursor coordinates and convert them to other coordinate system.

You may enable this component in two ways:

- From the Editor, clicking on the "Open Calculator" button at the bottom of the World Map inspector.
- From code, using any of its API through **map.calc** accessor. The first time you use its API, it will automatically add the component to the map gameObject.

On the Inspector you will see the following custom editor:



Converting coordinates from code

NEW! The asset provides a new **Conversion** static class which exposes many method to quickly convert between lat/lon and map position which don't require the calculator component. Example: **Conversion.GetLocalPositionFromLatLon**.

The calculator component API can be accessed as well from code through **map.calc** property. The conversión task involves 3 steps:

1. Specify the source unit (eg. "**map.calc.fromUnit = UNIT_TYPE.DecimalDegrees**").
2. Assign the source parameters (eg. "**map.calc.fromLatDec = -15.281**")
3. Call **map.calc.Convert()** method.
4. Obtain the results from the fields **map.calc.to*** (eg. "**map.calc.toLatDegrees**", "**map.calc.toLatMinutes**", …).

Note that the conversion will provide results for decimal degrees, degrees and plane coordinates. You don't have to specify the destination unit (that's only for the inspector window, in the API the conversion is done for the 3 types).

To convert from Decimal Degrees to any other unit you use:

map.calc.fromUnit = UNIT_TYPE.DecimalDegrees
map.calc.fromLatDec = <decimal degree for latitude>
map.calc.fromLonDec = <decimal degree for longitude>
map.calc.Convert()

To convert from Degrees, you do:

map.calc.fromUnit = UNIT_TYPE.Degrees
map.calc.fromLatDegrees = <degree for latitude>
map.calc.fromLatMinutes = <minutes for latitude>
map.calc.fromLatSeconds = <seconds for latitude>
map.calc.fromLonDegrees = <degree for longitude>
map.calc.fromLonMinutes = <minutes for longitude >
map.calc.fromLonSeconds = <seconds for longitude >
map.calc.Convert()

And finally to convert from X, Y (normalized) you use:

map.calc.fromUnit = UNIT_TYPE.PlaneCoordinates
map.calc.fromX = <X position in local sphere coordinates >
map.calc.fromY = <Y position in local sphere coordinates >
map.calc.Convert()

The results will be stored in (you pick what you need):

map.calc.toLatDec = <decimal degree for latitude>
map.calc.toLonDec = <decimal degree for longitude>
map.calc.toLatDegrees = <degree for latitude>
map.calc.toLatMinutes = <minutes for latitude>
map.calc.toLatSeconds = <seconds for latitude>
map.calc.toLonDegrees = <degree for longitude>
map.calc.toLonMinutes = <minutes for longitude >
map.calc.toLonSeconds = <seconds for longitude >
map.calc.toX = <X position in local sphere coordinates >
map.calc.toY = <Y position in local sphere coordinates >

You may also use the property **map.calc.captureCursor = true**, and that will continuously convert the current coordinates of the cursor (mouse) until it's set to false or you press the 'C' key.

map.calc.prettyCurrentLatLon returns the current latitude/longitude of the cursor location in a human readable format.

Using the distance calculator from code

The component includes the following two APIs to calculate the distances in meters between two coordinates (latitude/longitude) or two cities of the current selected catalogue.

map.calc.Distance(float latDec1, float lonDec1, float latDec2, float lonDec2)

map.calc.Distance(City city1, City city2)

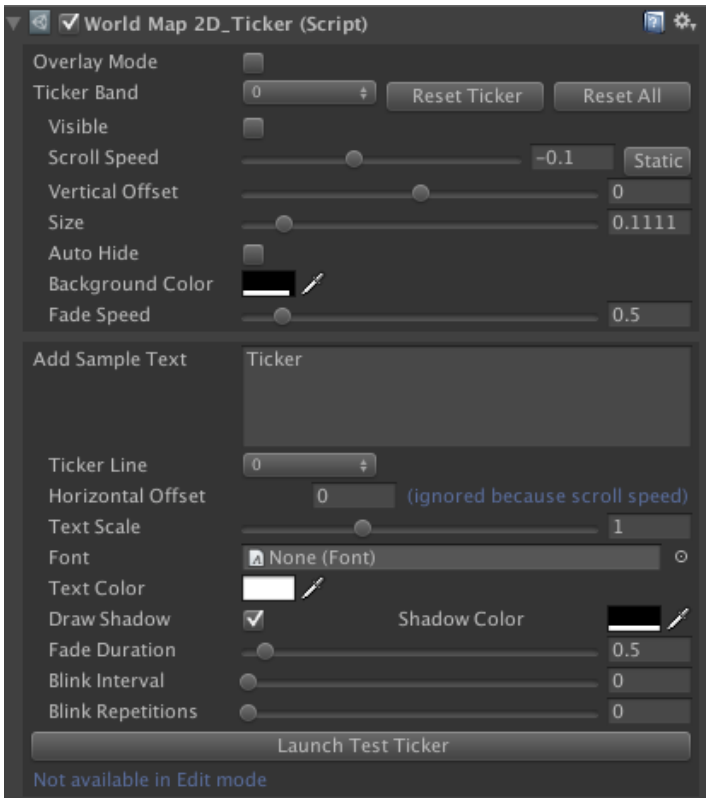
World Map Ticker

Use this component to show impact banners over the map. You can show different banners, each one with different look and effects. Also you can add any number of texts to any banner, and they will simply queue (if scrolling is enabled).

Similarly to the World Map Calculator component, you may enable this component in two ways:

- From the Editor, clicking on the "Open Ticker" button at the bottom of the World Map inspector.
- From code, using any of its API through **map.ticker** accessor. The first time you use its API, it will automatically add the component to the map gameObject.

On the Inspector you will see the following custom editor:



The top half of the inspector corresponds to the Ticker Bands configurator. You may customize the look&feel of the 9 available ticker bands (this number could be incremented if needed though). Notes:

- Ticker bands are where the ticker texts (second half of the inspector) scrolls or appears.
- A ticker band can be of two types: scrollable or static. You make a ticker band static setting its scroll speed to zero.
- Auto-hide will make the ticker band invisible when there're no remaining texts on the band.
- The fade speed controls how quickly should the band appear/disappear. Set it to zero to disable the fade effect.

It's important to note that everything you change on the inspector can be done using the API (more below).

In the second half of the inspector you can configure and create a sample ticker text. Notes:

- Horizontal offset allows you to control the horizontal position of the text (0 equals to zero longitude, being the range -0.5 to 0.5).
- Setting fade duration to zero will disable fading effect.
- Setting blink interval to zero will disable blinking and setting repetitions to zero will make the text blink forever.

The API can be accessed through map.ticker property and exposes the following methods/fields:

map.ticker.NUM_TICKERS: number of available bands (slots).

map.ticker.overlayMode: when enabled, ticker texts will be displayed on top of everything at nearclip distance of main camera.

map.ticker.tickerBands: array with the ticker bands objects. Modifying any of its properties has effect immediately.

map.ticker.GetTickerTextCount(): returns the number of ticker texts currently on the scene. When a ticker text scrolls outside the ticker band it's removed so it helps to determine if the ticker bands are empty.

map.ticker.GetTickerTextCount(tickerBandIndex): same but for one specific ticker band.

**map.ticker.GetTickerBandsActiveCount(**): returns the number of active (visible) ticker bands.

map.ticker.ResetTickerBands(): will reset all ticker bands to their default values and removes any ticker text they contain.

map.ticker.ResetTickerBand(tickerBandIndex): same but for an specific ticker band.

map.ticker.AddTickerText(tickerText object): adds one ticker text object to a ticker band. The ticker text object contains all the neccesary information.

The demo.cs script used in the Demo scene contains the following code showing how to use the API:

```
// Sample code to show how tickers work
void TickerSample() {
    map.ticker.ResetTickerBands();

    // Configure 1st ticker band: a red band in the northern hemisphere
    TickerBand tickerBand = map.ticker.tickerBands[0];
    tickerBand.verticalOffset = 0.2f;
    tickerBand.backgroundColor = new Color(1,0,0,0.9f);
    tickerBand.scrollSpeed = 0;    // static band
    tickerBand.visible = true;
    tickerBand.autoHide = true;

    // Prepare a static, blinking, text for the red band
    TickerText tickerText = new TickerText(0, "WARNING!!");
    tickerText.textColor = Color.yellow;
```

```
    tickerText.blinkInterval = 0.2f;
    tickerText.horizontalOffset = 0.1f;
    tickerText.duration = 10.0f;

    // Draw it!
    map.ticker.AddTickerText(tickerText);

    // Configure second ticker band (below the red band)
    tickerBand = map.ticker.tickerBands[1];
    tickerBand.verticalOffset = 0.1f;
    tickerBand.verticalSize = 0.05f;
    tickerBand.backgroundColor = new Color(0,0,1,0.9f);
    tickerBand.visible = true;
    tickerBand.autoHide = true;

    // Prepare a ticker text
    tickerText = new TickerText(1, "INCOMING MISSLE!!");
    tickerText.textColor = Color.white;

    // Draw it!
    map.ticker.AddTickerText(tickerText);
}
```

World Map Decorator

This component is used to decorate parts of the map. Current decorator version supports:

- ✓ Customizing the label of a country
- ✓ Colorize a country with a custom color
- ✓ Assign a texture to a country, with scale, offset and rotation options.

You may use this component in two ways:

- From the Editor, clicking on the "Open Decorator" button at the bottom of the World Map inspector.
- From code, using any of its API through **map.decorator** accessor. The first time you use its API, it will automatically add the component to the map gameObject.

The API of this component has several methods but the most important are:

map.decorator.SetCountryDecorator(int groupIndex, string countryName, CountryDecorator decorator)

This will assign a decorator to specified country. Decorators are objects that contains customization options and belong to one of the existing groups. This way you can enable/disable a group and all decorators of that group will be enabled/disabled at once (for instance, you may group several countries in the same group).

map.decorator.RemoveCountryDecorator(int groupIndex, string countryName)

This method will remove a decorator from the group and its effects will be removed.

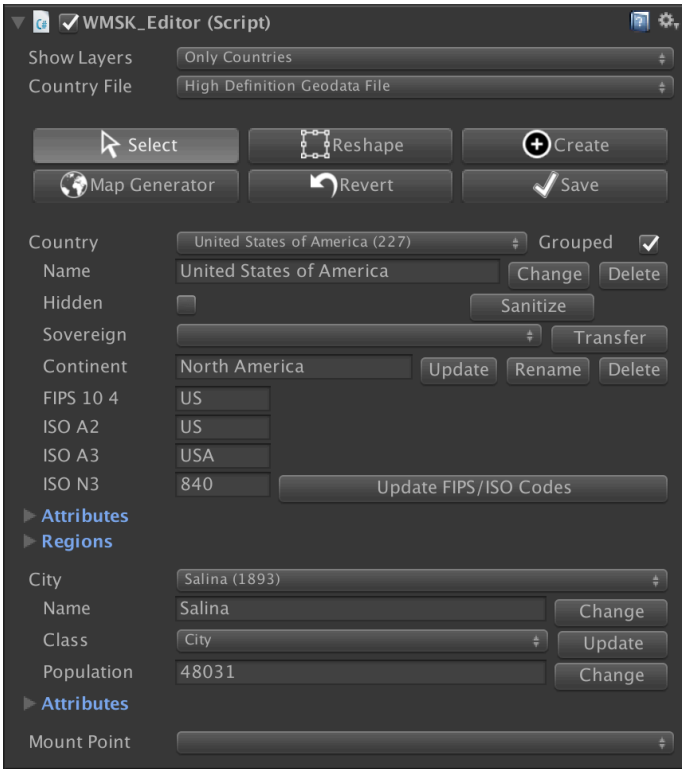
A decorator object has the following fields:

- **countryName**: the name of the country to be decorated. It will be assigned automatically when using SetCountryDecorator method.
- **hidden**: hides the country, including the region and label.
- **persistent**: when enabled, the decorator settings will be checked every 10th frames and will replace any other appearance change on the country.
- **customLabel**: leave it as "" to preserve current country label.
- **isColorized**: if the country is colorized.
- **fillColor**: the colorizing color.
- **labelOverridesColor**: if the color of the label is specified.
- **labelColor**: the color of the label.
- **labelVisible**: if the label is visible (default = true);
- **labelOffset**: horizontal and vertical position offset of the label relative to the center of the country.
- **labelRotation**: rotation of the label in degrees.
- **texture**: the texture to assign to the country.
- **textureScale**, **textureOffset** and **textureRotation** allows to tweak how the texture is mapped to the surface.

World Map Editor Component

Use this component to modify the provided maps interactively from Unity Editor (it doesn't work in play mode). To open the Map Editor, click on the "Open Editor" button at the bottom of the World Map inspector.

On the Inspector you will see the following custom editor:



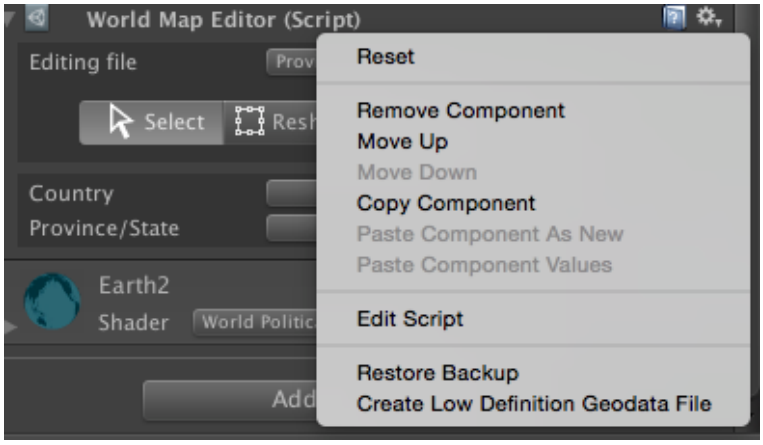
Description:

- **Show Layers**: choose whether to visualize countries or countries + provinces.hich layer to modify.
- **Country File**: choose which file to edit:
    - Low-definition geodata file (110m:1 scale)
    - High-definition geodata file (30m:1 scale)
- **City File**: choose which file to edit. Please note that changes in one file don't propagate to the other:
    - Principal city catalogue
    - Principal + medium sized cities catalogue
- **Country:** the currently selected country. You can change its name or "sell" it to another country clicking on transfer.
- **Create Background/Pool Country:** if not already created, a button on top of the provinces dropdown allows you to create a new special country with name "Pool". You can use this country as a default background country that can hold sea provinces and reassign the provinces later to any other country at will (check demo scene 106 under General Examples folder).
- **Province/State:** the currently selected province/state if provinces are visible (see Show Layers above). As with countries, you can change the province's name ore ven transfer it ot another country.
- **City:** the currently selected city.

Main toolbar

- **Select:** allows you to select any country, province or city in the Scene view. Just click over the map in Scene View. You can select multiple provinces holding Control key.
- **Reshape:** once you have either a country, province or city selected, you can apply modifications. These modifications are located under the Reshape mode (see below).
- **Create:** enable the creation of cities, provinces or countries.
- **Map Generation**: shows the map generator tool options.
- **Revert**: will discard changes and reload data from current files (in Resources/Geodata folder).
- **Save**: will save changes to files in Resources/Geodata foler.

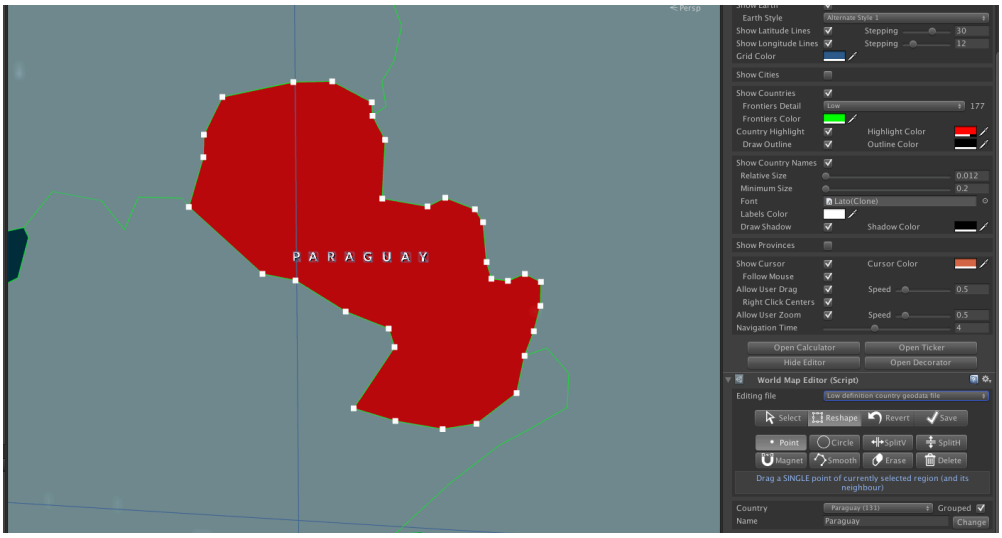
If you click the gear icon on the inspector title bar, you will see 2 additional options:



- **Restore Backup**: the first time you save changes to disk, a backup of the original geodata files will be performed. The backed up files are located in Backup folder inside the main asset folder. You may manually replace the contents of the Resources/Geodata folder by the Backup contents manually as well. This option do that for you.
- **Create Low Definition Geodata File:** this option is only available when the high-definition geodata file is active. It will automatically create a simplistic and reduced version (in terms of points) and replace the low-definition geodata file. This is useful only if you use the high-definition geodata file. If you only use the low-definition geodata file, then you may just change this map alone.

Reshaping options

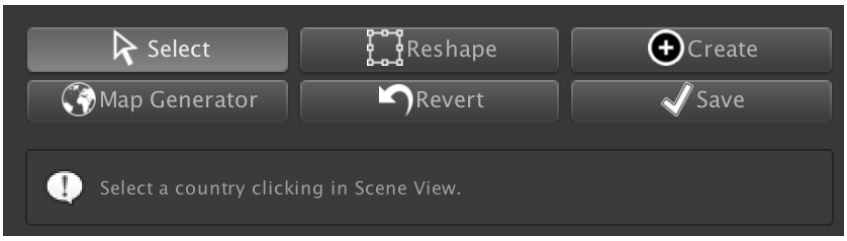
When you select a country, the Reshape main option will show the following tools:



- **Point tool**: will allow you to move one point at a time. Note that the corresponding point of the neighbour will also be moved along (if it exits). This way the frontier between two regions is easily modified in one step, avoiding gaps between adjacent regions.
- **Circle tool:** similar to the point tool but affects all points inside a circle. The width of the circle can be changed in the inspector. Note that points nearer to the center of the circle will move faster than the farther ones unless you check the “Constant Move” option below.
- **SplitV**: will split vertically the country or region. The splitted region will form a new country/province with name “New X” (X = original country name)
- **SplitH**: same but horizontally.
- **Magnet**: this useful works by clicking repeatedly over a group of points that belong to different regions. It will try to make them to join fixing the frontier. Note that there must be a sufficient number of free points so they can be fused. You can toggle on the option “Agressive Mode” which will move all points in the circle to the nearest points of other region and also will remove duplicates.
- **Smooth**: will create new points around the border of the selected region.
- **Erase**: will remove the points inside the selection circle.
- **Delete**: will delete selected region or if there’re no more regiosn in the current country or province, this will remove the entity completely (it disappear from the country /province array).

Create options

In “Create mode” you can add new cities, provinces or countries to the map:



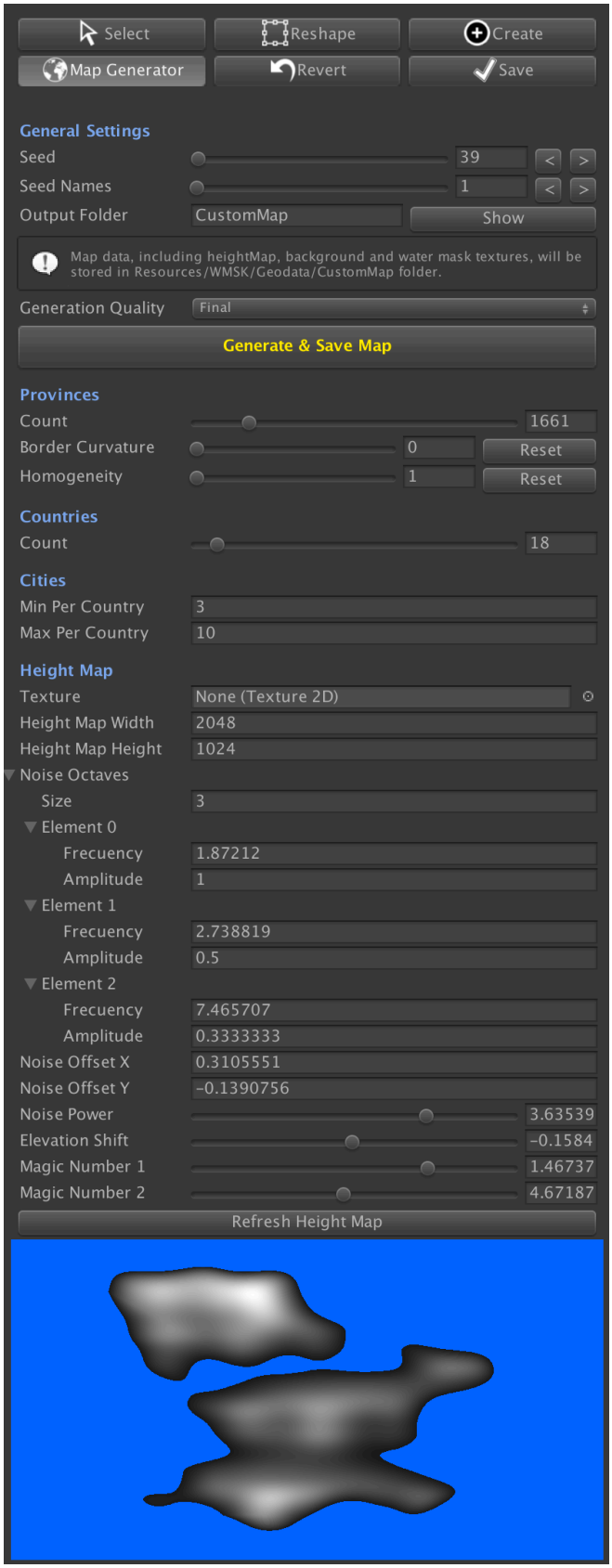
Note that a country is comprissed of one or more regions. Many countries have only one region, but those with islands or colonies have more than one. So you can add new regions to the selected country or create a new country. When you create a new country, the editor automatically creates the first / main region.

Also note that the main region of a country is the biggest one in terms of euclidean area.
Provinces have also regions and can have more than one.

You can also create new regions by splitting existing countries and provinces in two drawing a separation line between two border points. To perform the split start drawing a line near a border and continue cliking adding points and extending the line towards the opposite border (notice keyboard shortcuts printed in the Scene View window).

Map Generator

The Map Generator tool is designed to automate the process of creating an entire world map with a few clicks:



Use the "Seed" slider to quickly generate a different combination of values.

Press "Generate & Save Map" to create the world map files and update the representation. Choose "Draft" as generation quality while testing parameters as this option increases the speed of map generation.

Please note that when you click "Generate & Save Map", the tool will generate all geodata files and textures and put those files into the folder specified in the information text above the "Generation Quality" dropbox.

You can choose different sub-folders by using the "Output Folder" field hence you can use a different map for each scene in your game.

Once you click "Generate & Save Map" the new map will appear in the screen. Random names will also be generated:

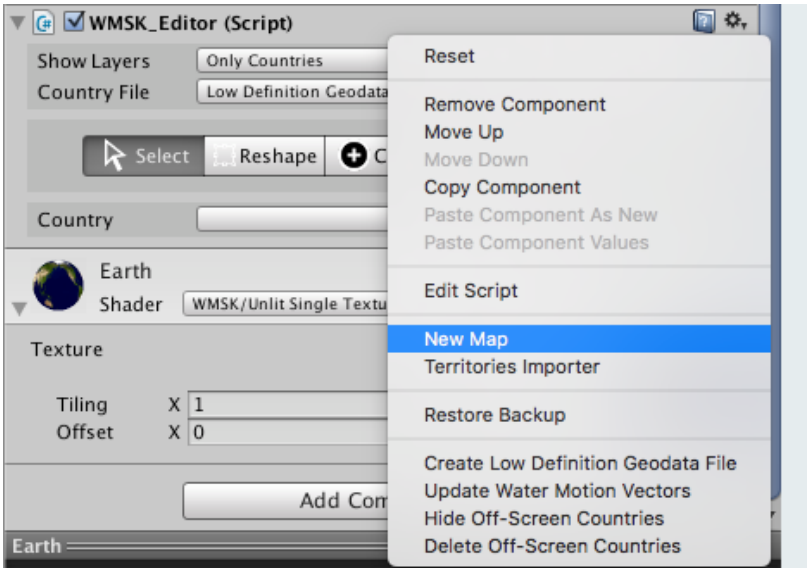
Creating a new world map from scratch step by step

The process of creating a new world map step by step includes:

1.- Clearing existing geodata information.
2.- Creating your own country frontiers, province borders and/or cities.
3.- Assigning your own world map textures.

Clearing existing geodata information

To clear current geographic data and start your own map from zero, use “New Map” option under Map Editor’s gear icon:



Creating your own country frontiers, province borders and/or cities

This task is the most complex. Fortunately WMSK provides you with a territory importer so you can automatically create rough frontiers for countries and provinces based on a color texture where each color represents a country (see Territory Importer section below).

Alternatively, or after using the Territory Importer, you can use the Map Editor to adjust frontiers and create your new cities.

Assigning your own world map textures

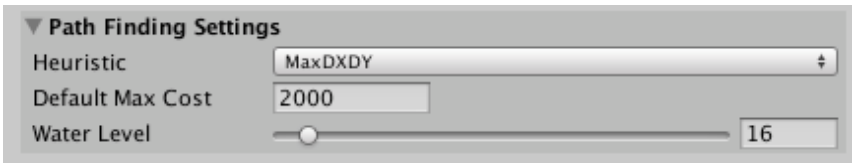
Depending on the Earth style you prefer, one or more textures need to be provided:

Main world texture for the background. Select the material of the Earth style and assign your own texture. The materials are located in WorldMapStrategyKit/Resources/WMSK/Materials folder.

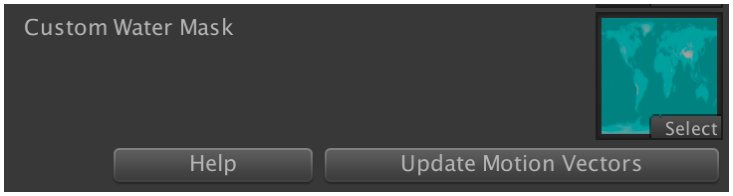
Heightmap and Watermask. If you use the Scenic styles and the viewport mode, you will also need to provide a suitable heightmap and watermask. Set them in the WMSK inspector in section Earth Settings:

The heightmap is required to calculate the elevation in the viewport mode. It should be a grayscale texture.

The water mask is similar to the heightmap with alpha channel used for specifying the water level in pathfinding and scenic animated foam. The water level can be configured in the Path Finding settings:

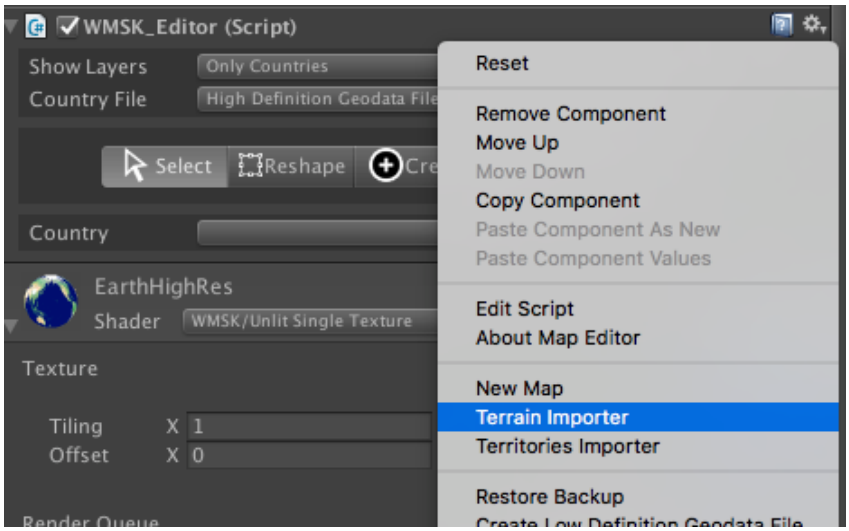


Once you assign a new water mask, it's important to generate the water motion vectors. The RGB channels of the watermask texture will contain water flow data used in Scenic Earth styles and viewport mode:

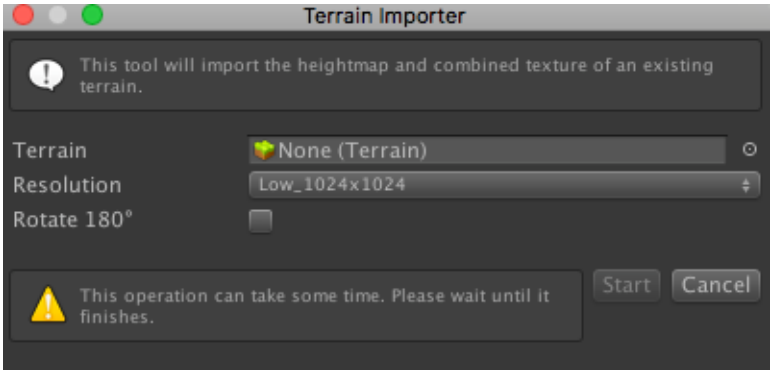


Terrain Importer

The Map Editor includes a terrain importer option located in the gear icon:



This tool can be used to import the heightmap and combined textures from a standard Unity terrain into WMSK. Clicking on the menu option brings the following dialog:

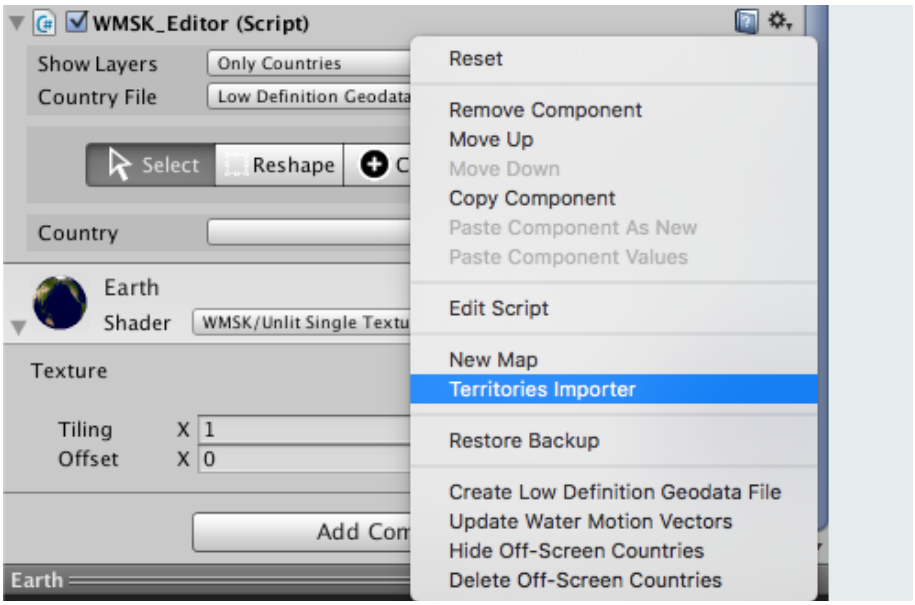


1.- Drag & drop a terrain from your scene.
2.- Select the texture resolution. Note that the heightmap always have a resolution of 2048x1024.
3.- Rotate 180 degrees. Tick this option if your terrain is rotated.

Once you click “Start”, the Earth style will be switched to “Texture” and the new texture and heightmap will be automatically assigned. These textures are stored in Resources/WMSK/Terrain folder.
You can also select other Earth styles and assign the new texture to the corresponding material found in Resources/WMSK/Materials.

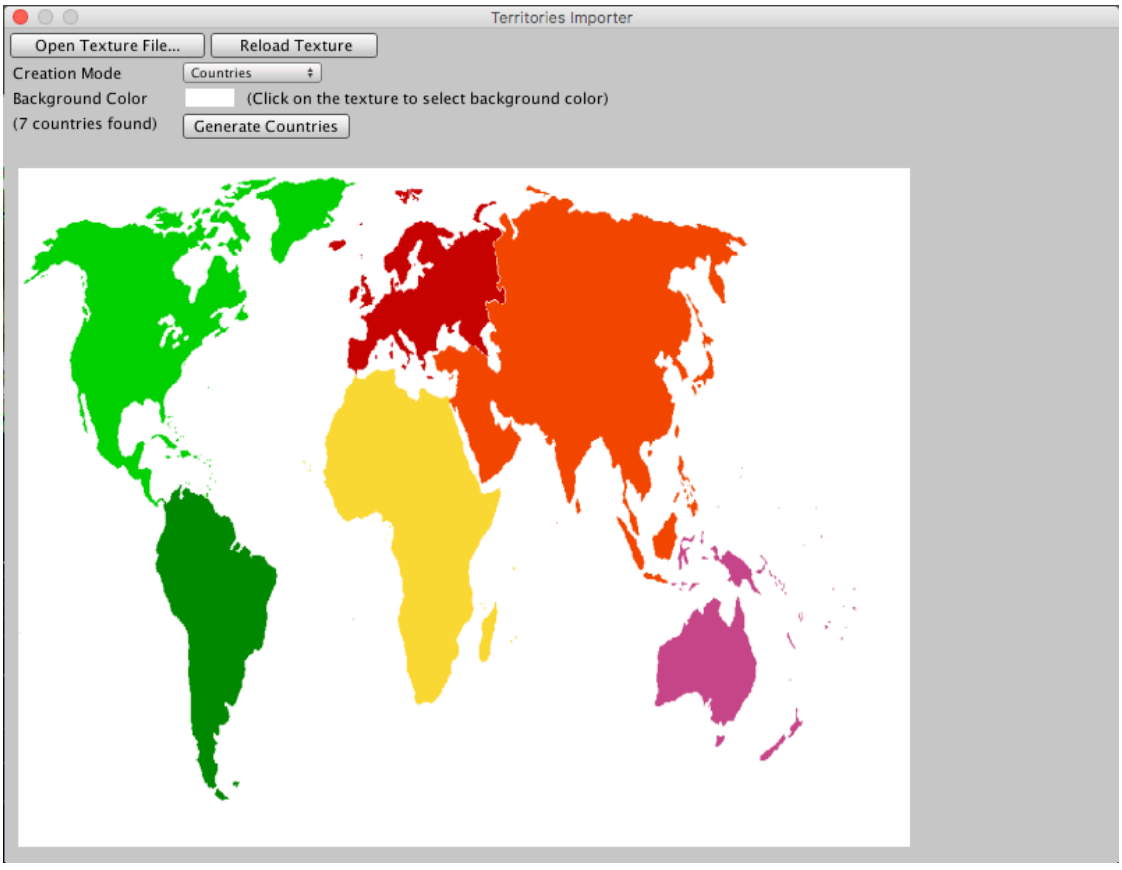
Territory Importer

The Map Editor includes a territory importer option located in the gear icon:



This tool can be used to speed up the creation of entire new world maps. It works by taking a previously prepared texture and creating country or province regions based on the colors of the texture.

The texture should have a 2x1 aspect ratio (example: 2048x1024 PNG file) and should contain solid colors only, like in the picture below:



Using territory importer is quite straight-forward:

1) Click “Open Texture-File” to select your texture file.
2) Select a background color (this color will be ignored). Territory Importer automatically takes the color for the first pixel but you can click on the texture to choose any other color as background.
3) Select the creation mode (countries or provinces).
4) Click “Generate Countries” or “Generate Provinces”.

From this point, the territory importer will start analyzing the texture and extracting regions.

Important: your current map will be replaced! You can use the texture importer safely as long as you don’t save the changes from within the Map Editor. So you can import the same texture (modified or retouched) as many times as you want. Click “Revert” in the Map Editor to cancel and reload map frontiers from current geodata files.

In Country Mode, the territory importer will create one country per different color found in the texture. Then it will add any region of the same color to each country. Finally it will update current provinces, cities and any created mountpoints to be associated to the new countries (for provinces, the center of the province will be checked against the new countries). Any province, city or mount point that don’t fit inside any of the new countries will be removed.

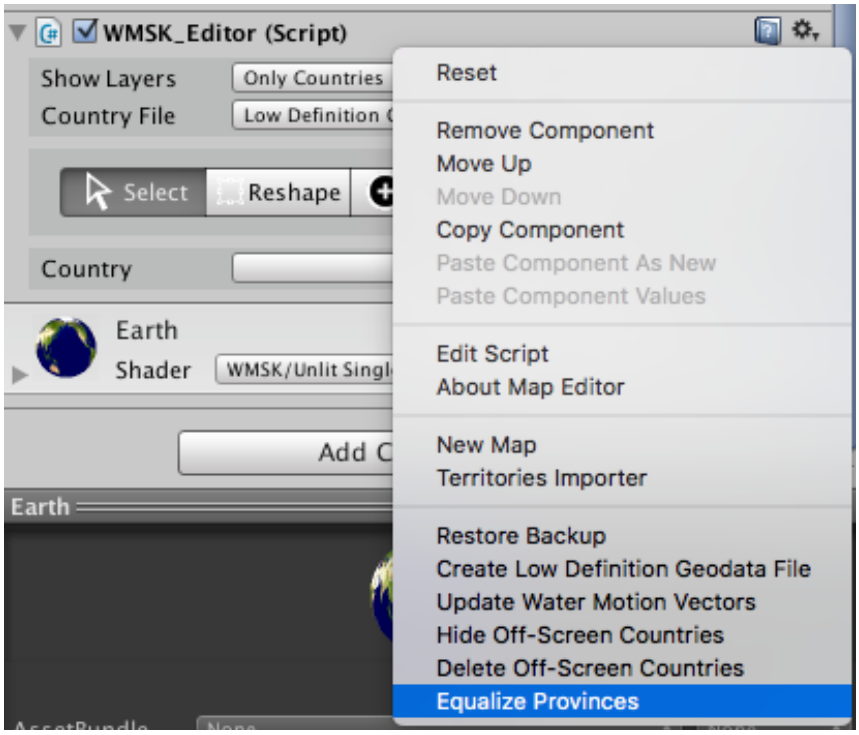
In Province Mode, the operation is very similar. Territory importer will create one province per different color and associate any region of same color to each province. Provinces usually have one region, but you could have one province split in 2 or more land regions this way. Once all provinces are created, they will be associated to the existing countries (again, the center of the province will be used to find a country that contains that point). Cities will be updated to reflect the new province they belong.

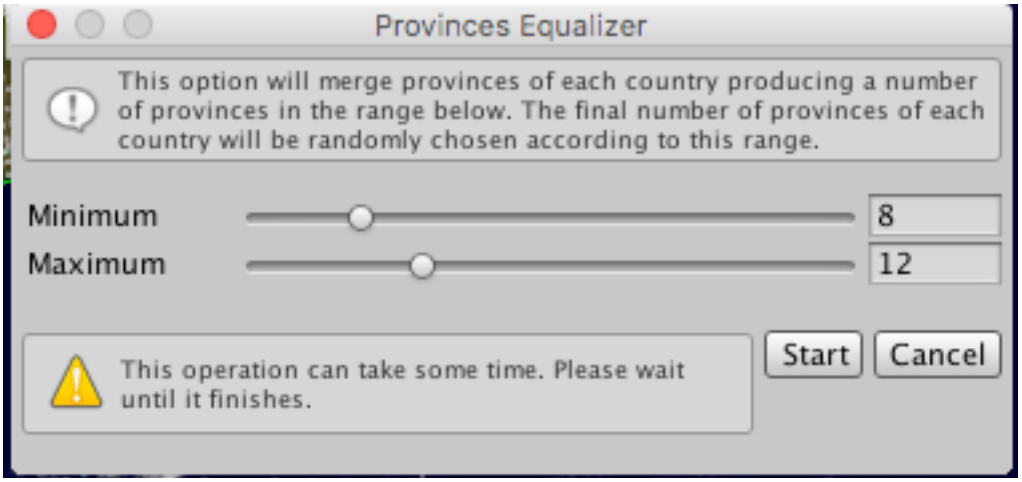
Note that resulting map will require some manual tweaks. Pay attention to the joints between frontiers or borders of two different regions. Use the magnet tool in the Map Editor to fix and merge frontier points.

Once you’re satisfied with the results, hit “Save” in the Map Editor to make changes persistent (it will overwrite current geodata files in Resources/Geodata folder).

Countries and Provinces Equalizer

The Map Editor includes a tool aimed to balance the number of countries per continent and provinces among countries. This tool is very useful for making games where the number of provinces should be similar between countries or in any case do not exceed certain number so the map does not show too cluttered.





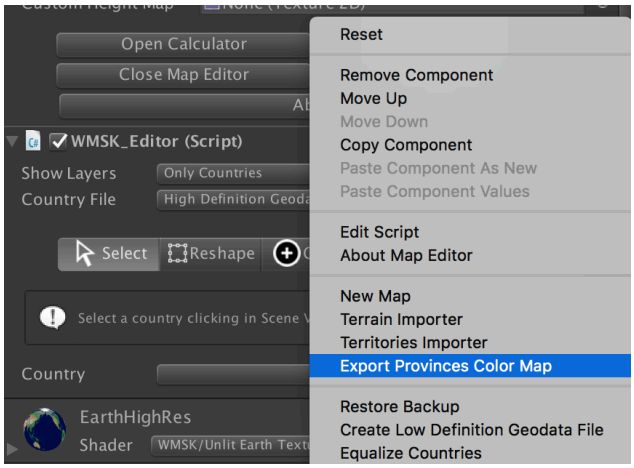
Important!

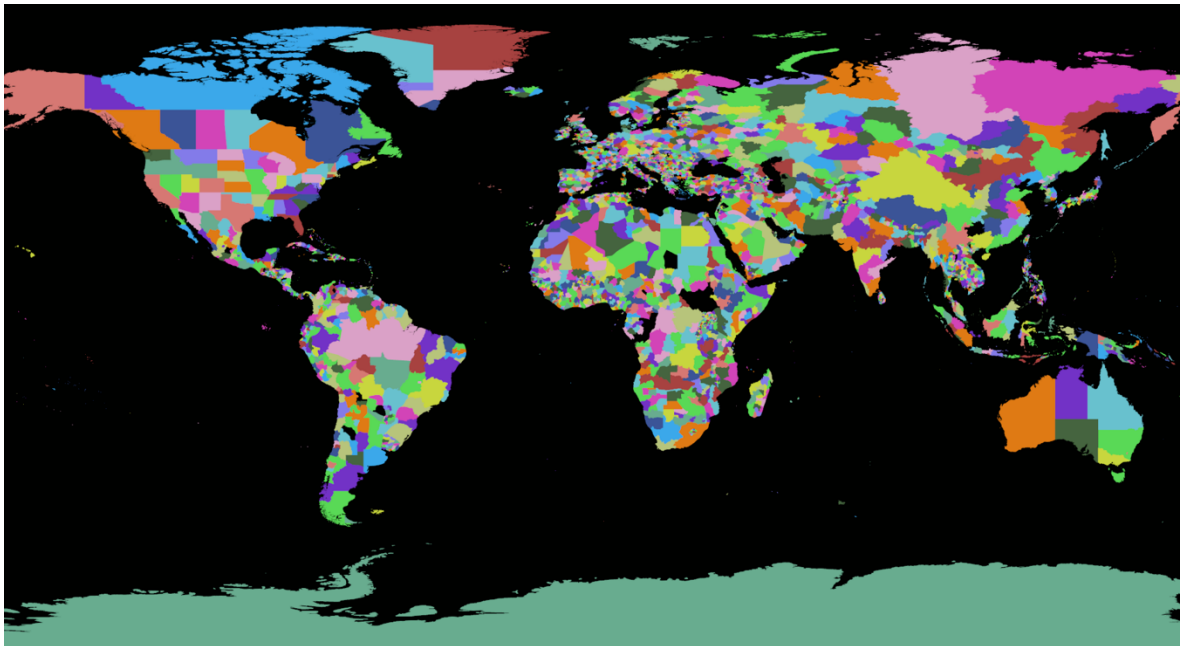
- Note that after executing this tool the new provinces are not saved to disk yet! You need to click "Save" to persists the changes before running your game or any change will be lost.

- Make a backup before saving these changes if you already have custom changes on the map (a single copy of the Geodata folder inside WorldMapStrategyKit/Resources/WMSK is enough). It's wise to have several copies of geodata folder in case you need/want to go back to certain point.

Export Provinces Color Map

Use this option to generate a big texture file with all provinces colored with random colors:





Then use the **ImportProvincesColorMap()** method to create new countries based on the province colors. Each unique color will produce a new country and all provinces sharing the same color will be remapped to that country.

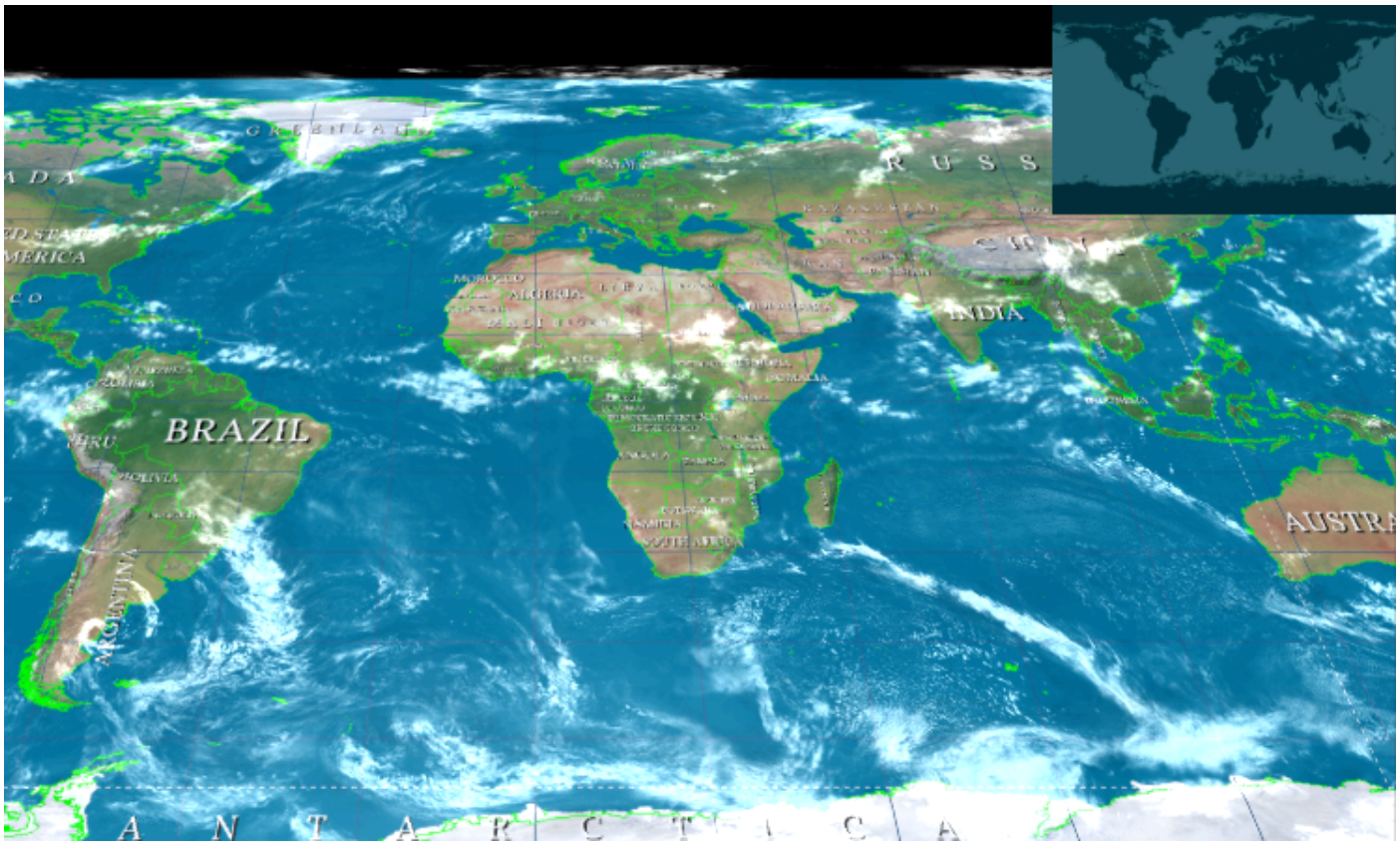
You can use an image editor like Gimp or Photoshop (or a custom C# script) to recolor the provinces. If you want to remove a province just color it with transparent color (alpha = 0).

Editor Tips

- **Before start making changes, determine if you need the high-definition file or not**. If you don't need it for your project, then you can just work with the low-definition file. Note that the high-def and low-def files are different. That means that changes to one file will not affect the other. This may duplicate your job, so it's important to decide if you want to modify both maps or only the low-def map.
- The high-definition file has lots of points. Current operation in this mode on some big countries (like Russia or Antartica) can be quite slow on some machines (although functional).
- You may change temporarily the scale of the map gameobject to 2000,1000,1 to make easier both the zoom and selection operations.
- If you decide to modify the high-definition file but also want to be able to use a low-definition version of the same map, you should complete all your modifications first in the high-definition map. Then use the command "Create Low Definition Geodata File" from the gear command, and adjust the low-def map afterwards.
- If you make any mistake using the Point/Circle tool, you can Undo (Control/Command + Z or Undo command from the Edit menu).
- Alternatively you can use the Revert option and this will reload the geodata files from disk (changes in memory wiil be lost).
- If you modified the geodata files in Resources/Geodata and want to recover original files, you can use the Restore Backup command from the gear icon, or manually replace the Resourcess/Geodata files with those in the Backup folder.
- As a last resort you may replace currnet files with the originals in the asset .unitypackage.
- It's safe to make manual backup copies of the Resources/Geodata files and replace them at any moment. They are just plain text files that are read by the asset during any initialization, which occurs when you hit play or make Unity rebuild your project.
- Beware of hitting play before saving your changes! If you run your game, all changes that have not been saved will be lost!
- Remember to visit us at kronnect.com for help and new updates.

MiniMap

World Map Strategy Kit also includes a minimap (see Demo scene #401 in UI Examples):



The minimap allows you quickly navigation to any location on the map.

To enable it, just call **WMSKMiniMap.Show(normalizedScreenPosition)**. This call will return a WMSKMiniMap reference which you can use to customize the minimap behaviour and appearance.

normalizedScreenPosition is a Vector4 which contains the normalized (0..1) screen coordinates and size for the minimap. Example:
Vector4 normalizedScreenPosition = new Vector4 (left, top, width, height) … where left, top, width and height are values in the range of 0..1
Note that screen position is 0,0, at bottom/left and 1,1 at top/right.

Static methods:

WMSKMiniMap.Show(normalizedScreenPosition): creates and show a minimap.

WMSKMiniMap.IsVisible(): returns true if minimap is currently visible.

WMSKMiniMap.RepositionAt(normalizedScreenPosition): moves the minimap to another position or changes its size.

WMSKMiniMap.Hide(): destroys the minimap.

Instance properties and methods:

The reference returned by WMSKMiniMap.Show() provides the following functionality:

Assuming **WMSKMiniMap minimap = WMSKMiniMap.Show(…)**:

mnimap.map: returns a reference to the map used as minimap, so you can customize its look and behaviour (it uses a normal WMSK map as well, so all properties applies here, like cursorColor and so on).

minimap.duration: the duration of the navigation when the user clicks over the minimap. Defaults to 2 seconds.

minimap.zoomlevel: the zoom level for the navigation target. Defaults to 0.1.

World Flags and Weather Symbols

This package is available as a separate purchase and includes +270 vector and raster images of country flags and weather symbols:

For more information, please consult the Asset Store page:
https://www.assetstore.unity3d.com/#!/content/69010

Once imported into the project, the names of the flag texture files equal to the country names used in our map assets so you can add flag icons or texture countries with their flag with minimal effort.

The code to texture the country surface with its flag would be:

```
// Get reference to the API
WMSK map = WMSK.instance;

// Choose a country
string countryName = “China”;

// Load texture for the country
Texture2D flagTexture = Resources.Load<Texture2D> ("Flags/png/" + countryName);

// Apply texture to the country main region (ignore islands)
int countryIndex = map.GetCountryIndex(countryName);
map.ToggleCountryMainRegionSurface(countryIndex, true, flagTexture);
```

Third-party support and integrations

NGUI

Integration with NGUI covers:

- Viewport mouse events (through PointTrigger script which is already attached to Viewport prefab).
- Mouse over GUI detection to prevent unwanted map interaction when cursor is over a GUI element (edit WMSKInternal.cs script file and uncomment the line at top:
  // #define NGUI_SUPPORT // Uncomment this line

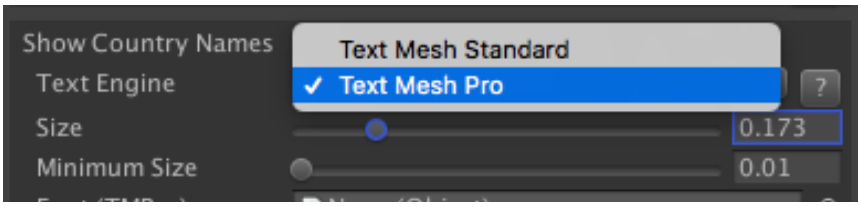
Note that only the viewport can be used as part of the NGUI hierarchy. The main object (WMSK) must not be added to NGUI and should be positioned in the scene independently of NGUI.

TextMesh Pro

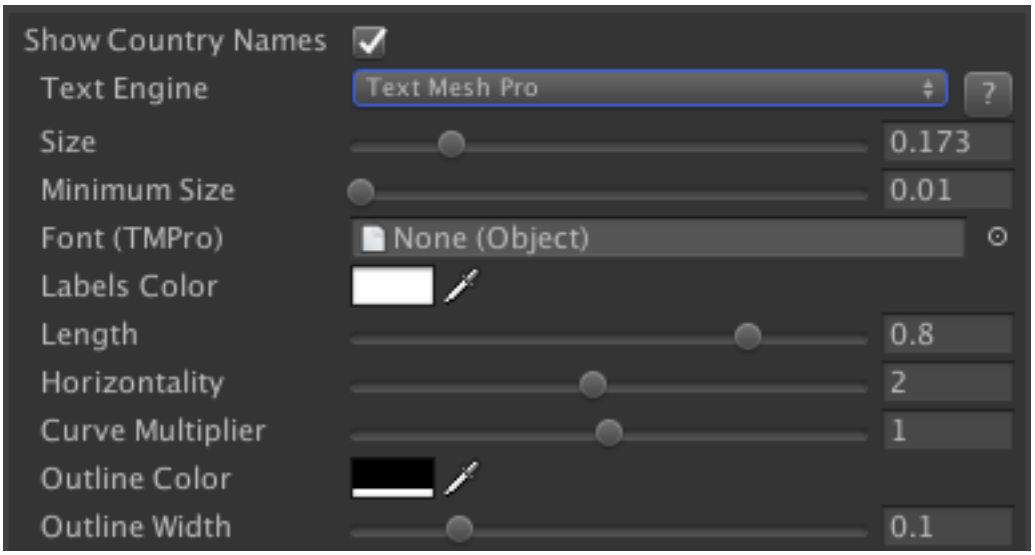
Integration with TextMesh Pro improves the appearance of labels:

- TextMesh Pro renders faster and better than standard TextMesh.
- Label placement is improved, including curved text along the country shape.

To enable Text Mesh Pro system, choose “TextMesh Pro” as the text rendering engine in the WMSK inspector:



The following options are specific to TextMesh Pro mode:



Size: the relative size for the label to the region size.

Minimum Size: makes smaller labels bigger.

Font (TMPro): the SDF font to be used. By default, WMSK uses the LATO SDF font asset located in WorldMapStrategyKit/Resources/WMSK/Font/TextMeshPro folder.

Labels Color: the color for the font face.

Length: specific scaling for the width of the label.

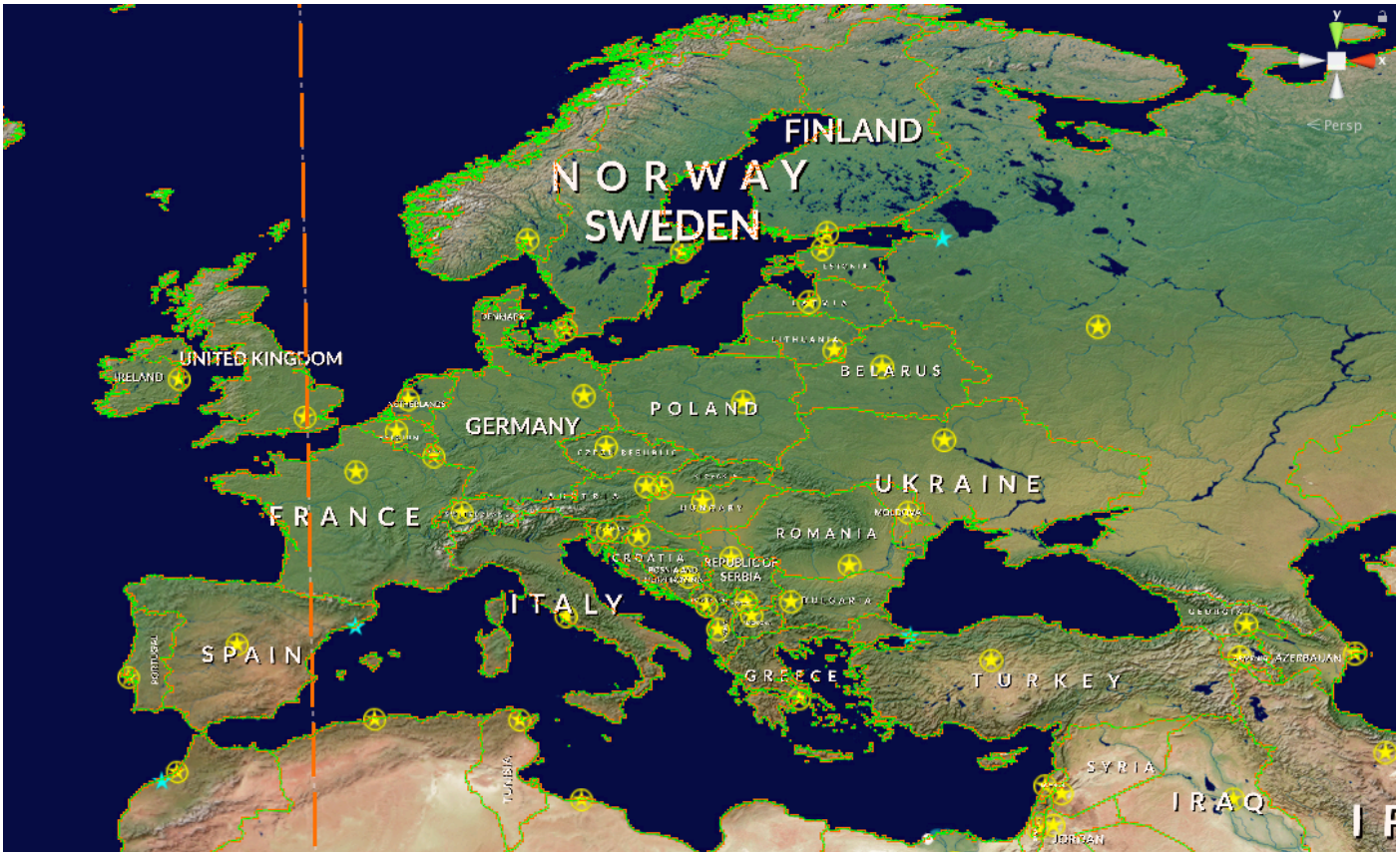
Horizontality: the greater the value, the more horizontal labels will be produced.

Curve Multiplier: a singular value which multiplies the curve applied to each label.

Outline Color/Width: attributes for the outline effect of the TextMesh Pro labels.

Here're 2 examples of labelling:

Standard Text Mesh labels:



Using Text Mesh Pro labels: