WORLD MAP STRATEGY KIT

WORLD MAP STRATEGY KIT MANUAL
V.1.3 / MAR.2016

Table of Contents

Introduction

Thank you for purchasing!

World Map Strategy Kit (WMSK) is a commercial asset for Unity 5.1.1 and above that allows to dramatically speed 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.
- 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.
- 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 (7 styles included)…
- 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.

QuickStart and Demo Scenes

1. Import the asset into your project or create an empty project.
2. You'll find several demo scenes inside the "Demos" folder. Start with the basic demos.
3. Run and experiment with the demonstration.

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#).

Support & Contact info

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

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

How to use the asset in your project

1. Drag the prefab “WorldMapStrategyKit” (WMSK) from “Resources/Prefabs” folder to your scene.
2. If you want to use the 3D surface / viewport (recommended), drag the prefab “Viewport” to the scene as well, and assign it to the WorldMapStrategyKit game object in the inspector.

Select the WMSK GameObject created to show custom properties:

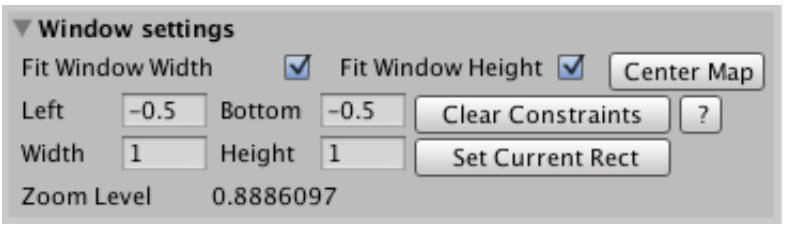
Custom Editor Properties

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



- **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.

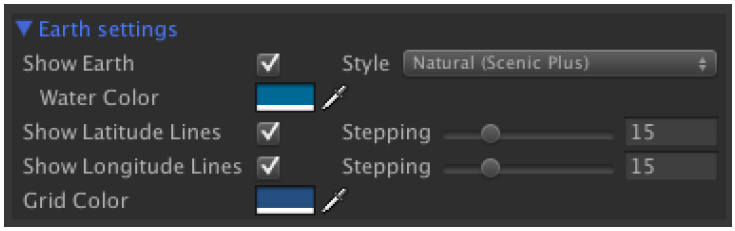
- **Ground Elevation:** sets the max. height for the surface when viewport is used.

- **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.

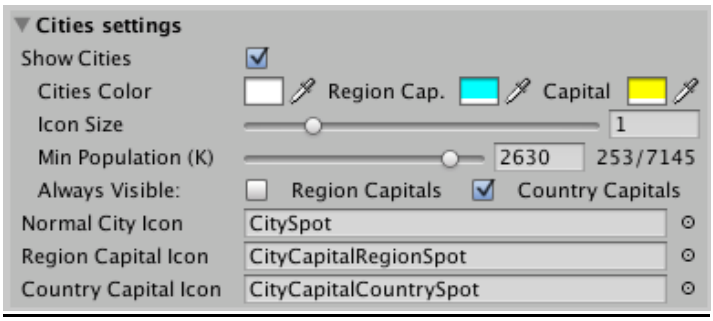
- **Style:** changes current style applied on the world map. Some styles can imply additional visual effects. See Using Scenic Styles section.

- **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).

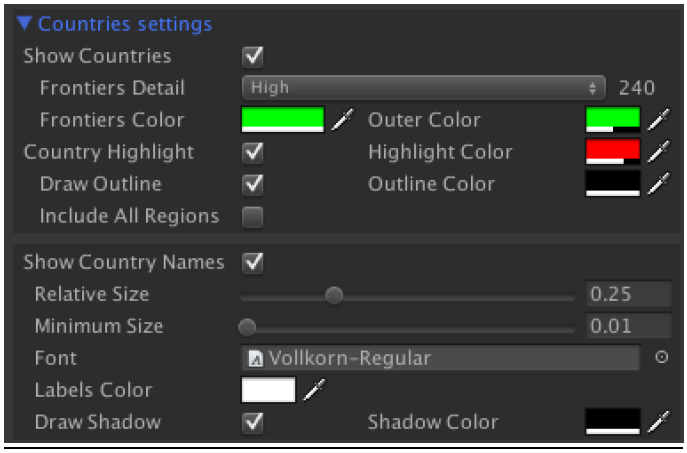
- **Icon Size**: scale multiplier for the cities icon.

- **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.

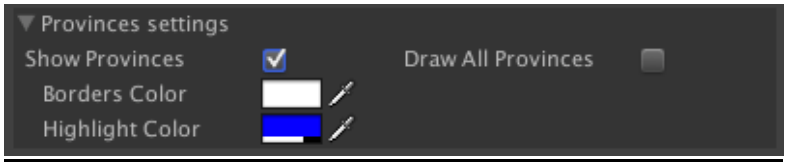
- **Show Country Names**: when enabled, country labels will be drawn and blended with the Earth map. This feature uses RenderTexture and has the following options:

- **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

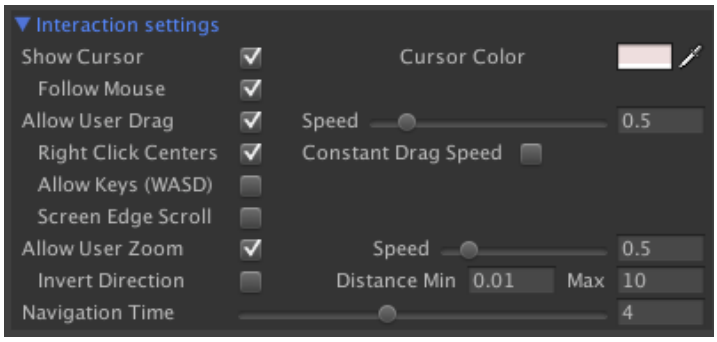


- **Show Provinces**: when enabled, individual provinces/states will be highlighted when mouse hovers them. Current active province can be determined using *provinceHighlighted* property (see API).

- **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.

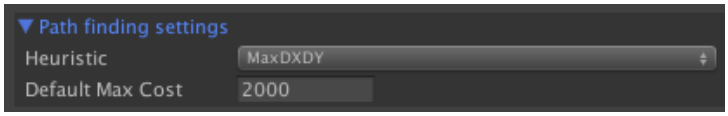
- **Allow User Drag:** as the title says, when enabled user can drag around the map with **Speed** parameter. You can also enable **Right Click Centers** which means that the map will scroll and center on the position the user right-clicks. **Contant Drag Speed** disables dragging acceleration **and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

Path Finding Settings



- **Heuristic:** defines the formula used when estimating the distance to the destination. It affects to the shape of the route. MaxDXDY produces more natural, human-like routes, while Manhattan will produce more zig-zag.

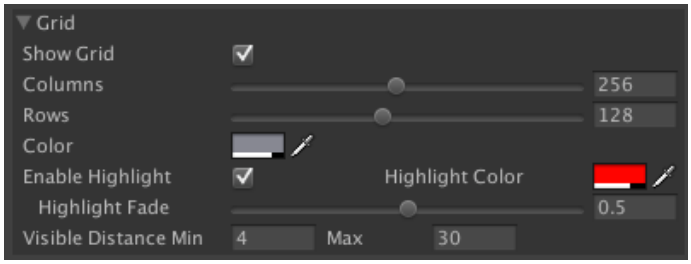
- **Default Max Cost**: this setting defined the maximum allowed cost of the route, meaning 1 point per horizontal/vertical movement and 2.64 points per diagonals.

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.

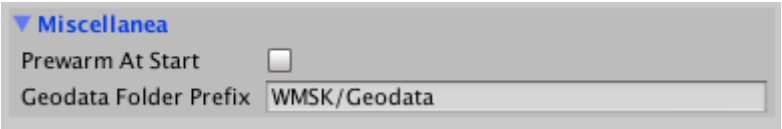
You can change these names to use different attribute sets.

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 feature

The asset allows to render the map inside a Viewport game object (provided in the asset as a Prefab).
The benefits for doing this are:

- To allow **cropping** the map inside the rectangle defined by the viewport gameobject area when panning or zoomin in.
- Render 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.

Adding your game objects to the viewport

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);**

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.

GO.**WMSK_MoveTo(Vector2 destination, float duration);**

Same than previous function except it accepts a Vector2 parameter.

GO.**WMSK_MoveTo(Vector2 destination, float duration, float height, HEIGHT_OFFSET_MODE heightOffsetMode, bool scaleOnZoom);**

This method will add the GO gameobject to the destination in duration seconds, but also it'll 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 camera follow the gameobject along its movement.

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.**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.

Additional options

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.

- **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.
- **destination**: the destination in map coordinates (-0.5, 0.5) of the current move.
- **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.
- **isMoving**: returns true if the gameobject is currently moving across the map.
- **currentMap2DLocation**: the current location of the game object in the map.
- **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.
- **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 zero will use the global setting defined by pathFindingMaxCost.
- **attrib**: JSON object that stores user-defined Custom Attributes (see Custom Attributes section for more details)

In addition to the properties above, 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.
- **OnMoveEnd(GameObjectAnimator):** fired when the game object stops.

Useful 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).

Using Path Finding feature with Game Objects added to the viewport

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 ground and water areas and another one for 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 call this property a copy of the current matrix is returned.

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 second matrix is a linearized 2D array of integers. 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 11 ("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 Scene MapPopulation, PathAndLines and PathFindingAdvanced for sample code.

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.

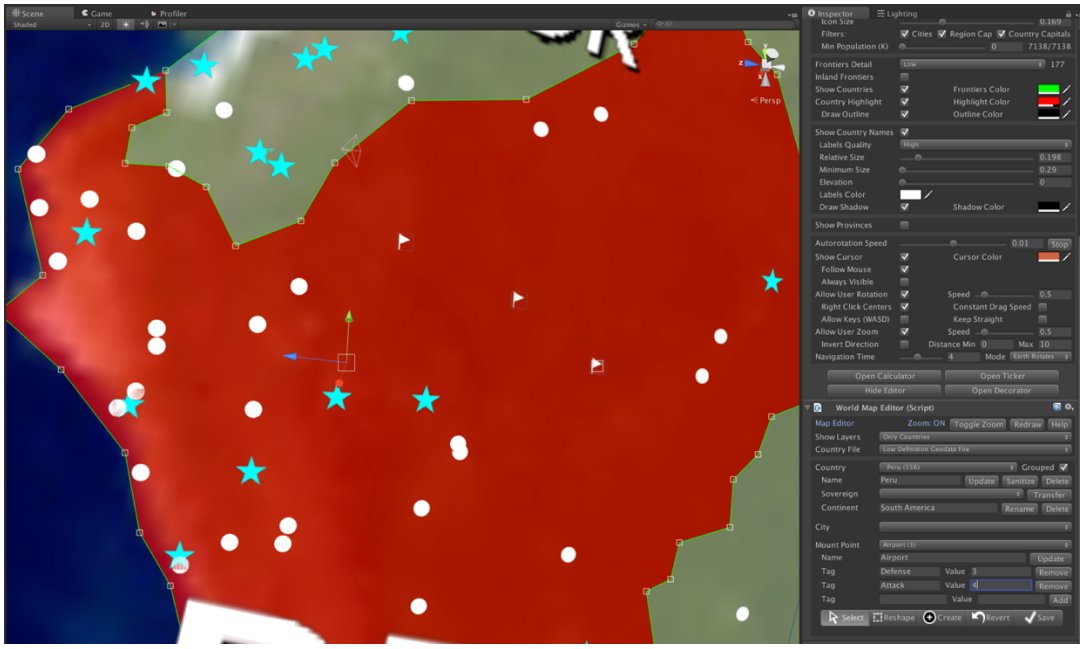
Mount Points

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

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 #8 covers all possible 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.GetCountriesXMLAttributes (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.GetCountriesXMLAttributes (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 SetCountriesXMLAttributes:

map.SetCountriesXMLAttributes(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.*

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 geometric center of the country in local space coordinates.
- **country.continent**: name of the continent to which the country belongs.

- **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**: root for custom attributes.

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.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**: root for custom attributes.

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.

Public Events

public event OnCountryEnter **OnCountryEnter**;
public event OnCountryExit **OnCountryExit**;
public event OnCountryClick **OnCountryClick**;

public event OnProvinceEnter **OnProvinceEnter**;
public event OnProvinceExit **OnProvinceExit**;
public event OnProvinceClick **OnProvinceClick**;

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

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

public event OnPathFindingCrossPosition **OnPathFindingCrossPosition**;

Public Properties & Methods

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

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

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

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

map.SetCountriesXMLAttributes(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.

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

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

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

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

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

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

map.SetProvincesXMLAttributes(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.

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

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

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

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 currently color used in the Earth when style = SolidColor.

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).

User 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 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.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.AddMarker(marker, planeLocation, markerScale): adds a custom marker (provided by you) to the map at plane location with specified scale (markerScale).

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.

map.AddCircle(position, kmRadius, ringWidthStart, ringWidthEnd, color): adds a circle to the markers overlay at map position, radius, ring size (0..1) and color (alpha supported).

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.

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.

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 value is 2000 cost points.

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.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.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.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.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.

Extra components accesors

map.calc: accesor to the Calculator component.

map.ticker: accesor to the Tickers component.

map.decorator: accesor to the Decorator component.

Additional Components

World Map Calculator

This component is useful to:

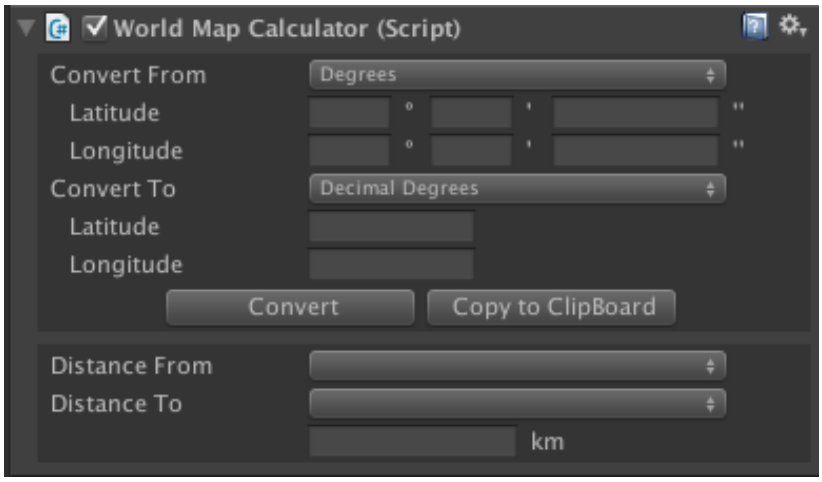
- Convert units from one coordinate system to another (for instance from plane coordinates to degrees and viceversa).
- Calculate the distance between cities.

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:



Using the converter from code

You may access the conversion API of this componet 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.

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)

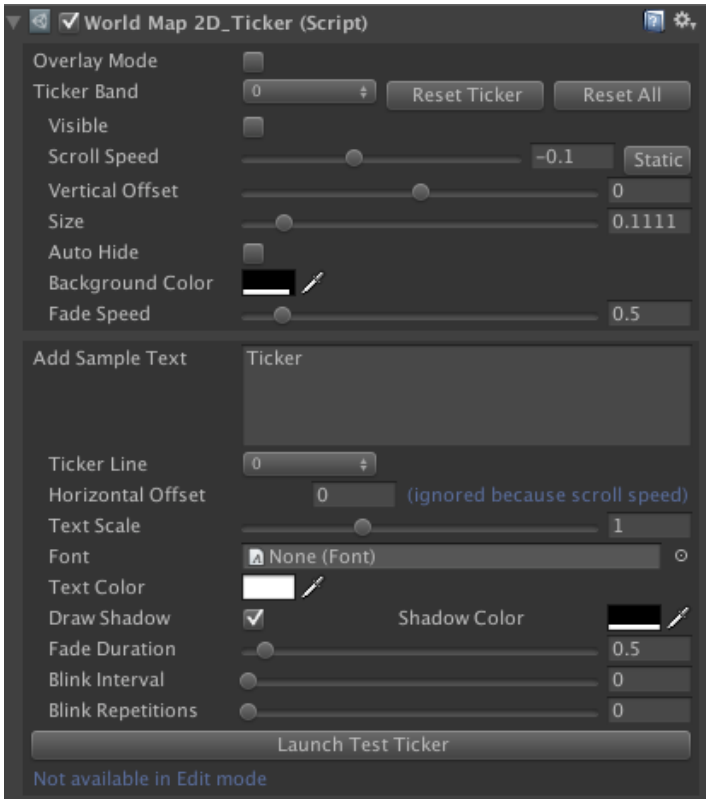
World Map Ticker

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.
- **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:

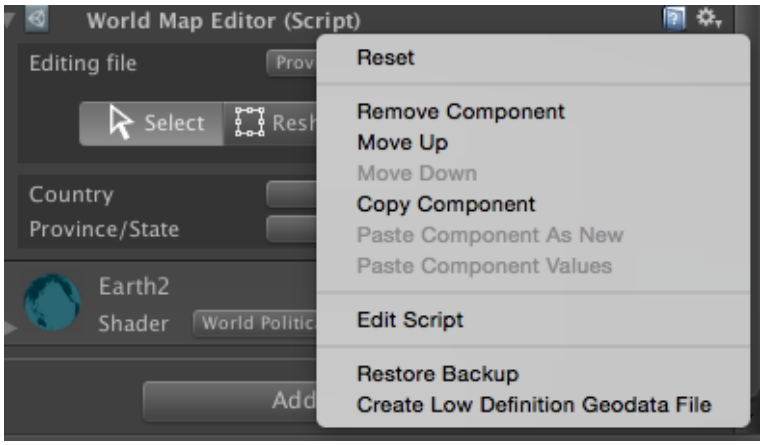
Description:

- **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.
- **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!
- **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.
- **Revert**: will discard changes and reload data from current files (in Resources/Geodata folder).
- **Save**: will save changes to files in Resources/Geodata foler.

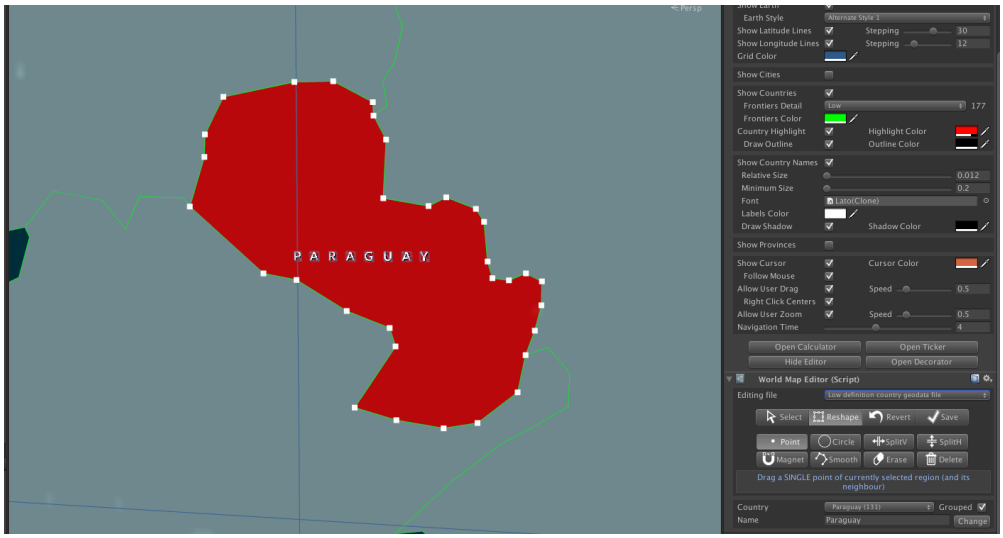
If you click the gear icon on the inspector title bar, you will see 2 additional options:



- **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

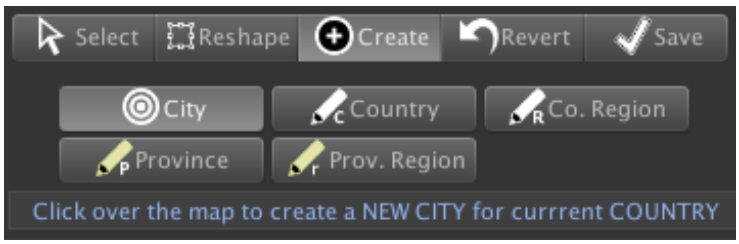
When you select a country, the Reshape main option will show the following tools:



- **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.

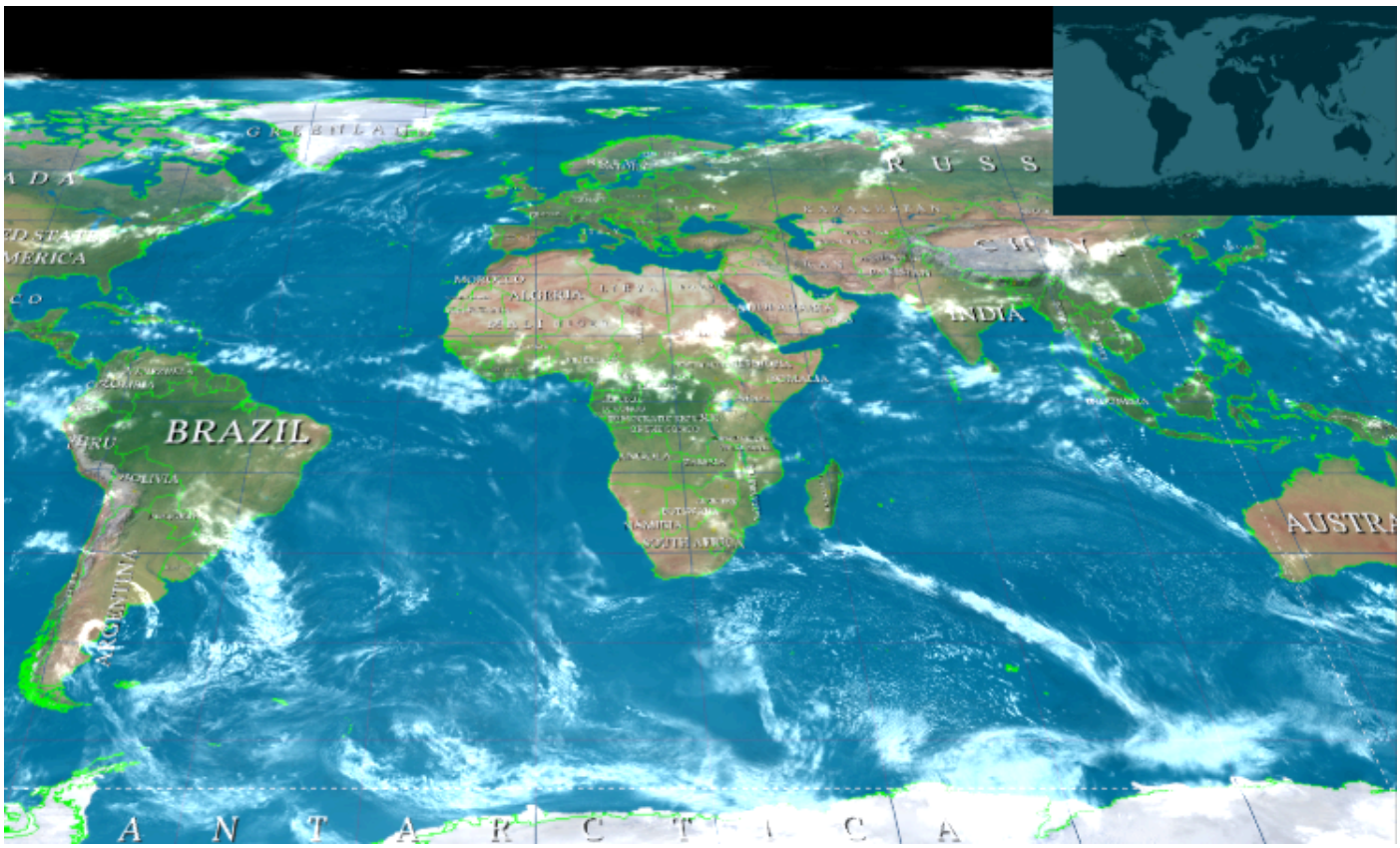
Editor Tips

- 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, you should complete all your modifications first in this 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.
- Remember to visit us at kronnect.com for help and new updates.

MiniMap

World Map Strategy Kit also includes a minimap (see Demo scene #7):



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.