# Coordinate Mapper

Coordinate Mapper is a free, open source project for visualizing datasets projected onto a globe in Unity. It gives you the flexibility to visualize data without having to set up custom models or adhere to specific JSON schemas without losing the ability to add any customization you need!

Also, Coordinate Mapper will do the spherical UV mapping for you, so no need for custom planet models or meshes.

## Installation

- Please download and install Coordinate Mapper from the Unity Asset Store: {Add link once available}
- Import JSON.Net for Unity into your project.
- Add a `Planet` layer to your project: `Edit > Project Settings > Tags and Layers`.

*Alternative*: if you don't wish to use the Asset Store, you can download the `Coordinate_Mapper.unitypackage` located in the `/meta/` folder. Inside of your unity project, right click on the *Assets* folder and choose `Import Package > Custom Package...`, then select the downloaded package to import.

## Initial Setup

**Add a planet model:** right click on the scene hierarchy and choose `Planet` or use the `Coordinate Mapper > Create Planet` menu item. There are 4 preset options: + **Lit Planet:** standard planet that respects lighting within the scene + **Unlit Planet:** same as Lit, but ignores lighting within the scene + **Lit Overlay Planet:** planet that contains 2 textures, one for the planet texture itself and another that gets overlaid using the alpha (used for heatmaps), and respects lighting within the scene + **Unlit Overlay Planet:** same as Lit Overlay, but ignores lighting within the scene

**Include a Planet layer:** In order for the plotting to work properly, you must add a `Planet` layer to your project, and set the created planet to use that layer. You can add a layer by going to `Edit > Project Settings > Tags and Layers`.

**Note**: In order to use any of the Coordinate Mapper scripts in code you must include the namespace by adding `using CoordinateMapper` to the top of your script.

## Visualization Options

Coordinate Mapper provides two different paths to visualizing data, a default visualizer and a custom visualizer. - The default implementation is for users who wants to see their data visualized and don't need to adjust or add anything

to the process. In order to use the default visualization all you need is data that contains latitude and longitude keys formatted in accordance with any of the supported formats (Outline in *KeyFormat* below). - The custom visualization is for users who have additional needs in terms of properties used to map points or visual adjustments.

### Default Visualization

**Add the visualizer script**: Once you have setup your planet, select it in the scene view and add the `Default Visualizer` script. This script will take a dataset, parse it, and plot the points using the default implementations provided with the framework. The supported schemas are outlined below:

**Properties:**

- *Data File:* The file to parse containing the coordinate data
- *Point Prefab:* The model that gets plotted on to the planet (supplied prefabs are inside the *Prefabs* folder)
- *Key Format:* How the latitude and longitude keys are stored within the dataset:
  - Json Lat and Lng Keys - separate latitude and longitude keys for each location
  - Json Single Lat Lng Array - array of alternating latitude and longitude values for locations
  - Json Lat Lng Array - separate arrays for latitude and longitude for locations
  - Csv - separate columns for latitude and longitude
- *Latitude Key:* The key within the dataset containing the latitude values
- *Longitude Key:* The key within the dataset containing the longitude values

The default visualizer also supports an **optional** magnitude key, which will scale the Z value of the point prefab for the location by the magnitude value. (See Earthquake Demo.)

- *Magnitude Key:* The key within the dataset containing the magnitude values
- *Load Complete:* An event that can be used to pass along the plotted points if any other script wants access to them

Just fill out the appropriate properties and press Play! You should see your data mapped to the planet!

### Custom Visualization

While the default visualizer is great to get something displaying quickly, it is purposefully limited in its use. If you require any extra keys other than *Latitude*, *Longitude*, and *Magnitude*, or if you want to customize the way GameObjects get mapped on to the planet, etc. . . , you will need to take the customized approach.

The custom approach can be broken out into two processes: parsing your data, and plotting your data.

**Parsing Your Data: IDataLoader:** extend this interface on any custom scripts you need to parse your datasets. This is useful if you need to parse out more values than the default visualizer allows (*Latitude*, *Longitude*, and *Magnitude*). IDataLoader ensures your class has the following: - *TextAsset dataFile:* the file containing your data (should be json or csv format) - *void ParseFile(string fileText):* should be used to take in the string representation of the dataset and to parse out and serialize any needed values - *DataLoadedEvent loadComplete:* a unity event for passing along the plotted coordinate points (will most likely be called at the end of the `ParseFile` function, if needed)

Scripts implementing this protocol should accept a file (*dataFile*) that you send to the appropriate parser through *ParseFile*, which returns the necessary info to create your *ICoordinatePoint* objects and sends them along (using *loadComplete*) to any scripts that care.

**Parsers**

There are two parser scripts, `JsonParser` and `CsvParser`, each taking its respective file type.

The `JsonParser` *Parse* functions accepts: - *string json* - the text from the file - *string[] keys* - the keys to parse out of the json text - Returns a `Dictionary<string, object[]>`, where each key sent along to the function is used as the key for the parsed values object[].

```
var jsonParsed = JsonParser.Parse(fileText,
    new string[] { "latitude", "longitude", "magnitude", "place", "time" });
```

The `CsvParser` *Parse* function accepts: - *string text* - the text from the file - Returns a `Dictionary<string, object[]>`, where the names for each column are used as the keys for the parsed values object[]

**Note:** The dictionary values are returned as an `object[]`. So it is up to you to convert them to the proper type. For example, if you were dealing with your longitude values, and they are floats in your dataset, you might cast them like this:

```
var lngs = jsonParsed["longitude"].Select(m => Convert.ToSingle(m)).ToArray();
```

Once you have everything properly parsed and casted, you can use the result to create and plot **ICoordinatePoint**s

**ICoordinatePoint:** extend this interface on the actual script that represents what should get plotted onto the planet. ICoordinatePoint ensures your class has the following: - *Location location:* class holding *latitude*, *longitude*, and *name* for coordinate point - *GameObject pointPrefab:* the model that gets plotted on to the planet - *GameObject Plot(Transform planet, Transform container,*

*int layer):* should be used to plot the prefab itself, and then to perform any custom manipulation needed on it (such as scaling): - *planet:* the transform of the GameObject that the point prefab should map to - *container:* the transform of the GameObject to add as a parent of the plotted point prefab - *layer:* the layer from *Tags and Layers* to use for the plotted point prefab

The main point of this protocol is to call the `Plot(Transform planet, Transform container, int layer)` function and handle the placement of the point to the planet. Typically, you would start by calling `PlanetUtility.PlacePoint()`, which returns the plotted GameObject. Once you have the GameObject, you can make any adjustments you need.

Here is an example from the Earthquake Demo:

```
public GameObject Plot(Transform planet, Transform container, int layer) {
    //Plot the point
    var plotted = PlanetUtility.PlacePoint(planet, container, location, pointPrefab);

    if (plotted != null) {
        plotted.layer = layer; //Set the layer

        //Add a Monobehaviour and store this class on it
        var eqPoint = plotted.AddComponent<EarthquakePoint>();
        eqPoint.info = this;

        //Adjust the scaling as a function of magnitude
        plotted.transform.localScale = new Vector3(plotted.transform.localScale.x,
            plotted.transform.localScale.y,
            plotted.transform.localScale.z * Mathf.Pow(2f, magnitude));

        //Orient the point to the surface of the planet at this location
        var point = (plotted.transform.position - planet.transform.position).normalized;
        plotted.transform.rotation = Quaternion.LookRotation(point);
    }

    return plotted;
}
```

**Heatmap:** can be added to the planet GameObject in order to visualize the data as a heatmap. Contains the following properties: - *M Planet Radius:* this is the planet's **radius** (not diameter) in meters (i.e., The Earth is 6371000) - *Km Range:* the range, in kilometers, each points of data should affect on the heatmap - *Start Value:* the value (between 0 and 100) that is added to the heatmap at the location of the data point - *End Value:* as we get farther from the location of the data point (up to the *Km Range*), the amount the point affects the location diminishes. *End Value* is how much effect should be added to the heatmap at the limit of *Km Range* (should be less than *Start Value*) - *Heatmap Size:* the size of the heatmap texture (in pixels) to use. (Keep in mind, the larger the

texture, the more processing intensive the heatmap) - *Colors:* the color gradient used to draw the heatmap (Gradient location 0 corresponds to heatmap value 0 and so on to gradient location 100) - *Hm Renderer:* the GameObject containing the Mesh Renderer with the proper *Overlay* material (described above)

There are two steps to generating a heatmap: - Ensure you have a material on your planet that can accept and display a secondary texture. There are a number of premade materials for this located at `Coordinate_Mapping > Materials`, they have *Overlay* appended to the name. Also, the Overlay versions of the planets from the *Add a planet* section outlined above come setup to accept a heatmap. - Call `public async void GenerateHeatmapGrid(IEnumerable<ICoordinatePoint> points)`. This accepts a list of *ICoordinatePoint*s, generates a heatmap texture from them, and supplies it to the material as the overlay texture (using the property reference name *_OverlayTex*.

The typical path for generating a heatmap would be to add the `Heatmap` script to your GameObject, and then to add the `GenerateHeatmapGrid` function as an event listener for your `IDataLoader`'s `loadComplete` event (Which sends along a list of `ICoordinatePoint`s).

Here is an example from the Earthquake demo:

**Note:** The heatmap only takes the *frequency* of points at a location into account. The more data points in a given area, the "hotter" the location. It doesn't take additional parameters into account to modify the weight that each plot gives to the heatmap.

## Demos

There are two demo projects included within the package to demonstrate how certain implementations work:

### Earthquake Demo

(located under `Coordinate_Mapping > Demos > Earthquake_Demo_Scene`)

This demo uses data for all earthquakes that happened around the world from 01/01/2019 to 01/01/2020 with a magnitude of 3.0 or greater. The dataset was obtained from the U.S. Geological Survey website.

Running the demo loads and plots the earthquake data, scaling the points based on their magnitude. It also generates a heatmap for the currently shown earthquake points.

There is a double sided slider in the top left you can use to limit the magnitude range of earthquakes plotted; as you adjust that range, the heatmap regenerates. You can use the mouse to rotate the camera around the Earth, as well as use the `z` and `x` keys to zoom the camera in and out.

Pressing the spacebar with the cursor hovering over an earthquake point will display a text box with additional information about the earthquake, and pressing it with the cursor outside of an earthquake point will hide the box.

If you wish to just look at the heatmap, there is a `Hide points` check-box that allows you to toggle the visibility of the earthquake points.

The space skybox in the background is from the asset store package SpaceSkies Free

#### Real-time Flights Demo

(located under `Coordinate_Mapping > Demos > Realtime_Flights_Demo`)

This demo uses The OpenSky Network API to pull real-time data about current flights around the world and plot them over the globe.

Running the demo will hit the OpenSky API (sometimes the response can take a number of seconds) and grabs the first 500 flights returned. It then plots them to the Earth, taking into account their altitude, heading, and velocity. At this scale, the altitude difference between planes isn't noticeable, so I apply a multiplier to exaggerate it.

We then hit the API 10 seconds after each response, and update the information applied to the planes in order to keep them properly situated as they fly.

Pressing the spacebar with the cursor hovering over a plane will display a text box with additional information about that flight and pressing it with the cursor off of a plane will hide the box. You can use the mouse to rotate the camera around the Earth, as well as use the `z` and `x` keys to zoom the camera in and out.

## License