# Technical documentation. - NORDARK DT

* 1 [Theory](#Technicaldocumentation-Theory)
  + 1.1 [Relation between an IES file and Unity lights](#Technicaldocumentation-Relationbetweena)
  + 1.2 [Luminance computation](#Technicaldocumentation-Luminancecomputa)
* 2 [Source Control](#Technicaldocumentation-SourceControl)
* 3 [Implementation](#Technicaldocumentation-Implementation)
  + 3.1 [Assets organization](#Technicaldocumentation-Assetsorganizati)
  + 3.2 [Global information on scripts](#Technicaldocumentation-Globalinformatio)
  + 3.3 [Sky (sun and moon)](#Technicaldocumentation-Sky(sunandmoon))
  + 3.4 [EventSystem](#Technicaldocumentation-EventSystem)
  + 3.5 [Main camera](#Technicaldocumentation-Maincamera)
  + 3.6 [Global volume and HDRP](#Technicaldocumentation-GlobalvolumeandH)
  + 3.7 [Terrain](#Technicaldocumentation-Terrain)
  + 3.8 [Mapbox](#Technicaldocumentation-Mapbox)
  + 3.9 [Vegetation Studio Pro](#Technicaldocumentation-VegetationStudio)
  + 3.10 [UI](#Technicaldocumentation-UI)
  + 3.11 [Managers](#Technicaldocumentation-Managers)
    - 3.11.1 [IES Manager](#Technicaldocumentation-IESManager)
    - 3.11.2 [Light Poles manager](#Technicaldocumentation-LightPolesmanage)
    - 3.11.3 [Cameras manager](#Technicaldocumentation-Camerasmanager)
    - 3.11.4 [Vegetation objects manager](#Technicaldocumentation-Vegetationobject)
    - 3.11.5 [Locations manager](#Technicaldocumentation-Locationsmanager)
    - 3.11.6 [Ground Textures manager](#Technicaldocumentation-GroundTexturesma)
    - 3.11.7 [Data visualization manager](#Technicaldocumentation-Datavisualizatio)
    - 3.11.8 [Game manager](#Technicaldocumentation-Gamemanager)
    - 3.11.9 [Scene manager](#Technicaldocumentation-Scenemanager)
    - 3.11.10 [Street view manager](#Technicaldocumentation-Streetviewmanage)
    - 3.11.11 [Luminance map manager](#Technicaldocumentation-Luminancemapmana)
    - 3.11.12 [Light computation manager](#Technicaldocumentation-Lightcomputation)
    - 3.11.13 [Light configurations manager](#Technicaldocumentation-Lightconfigurati)
    - 3.11.14 [Weather manager](#Technicaldocumentation-Weathermanager)
    - 3.11.15 [Measure manager](#Technicaldocumentation-Measuremanager)
  + 3.12 [Splash screen](#Technicaldocumentation-Splashscreen)
  + 3.13 [Assets](#Technicaldocumentation-Assets)

# Theory

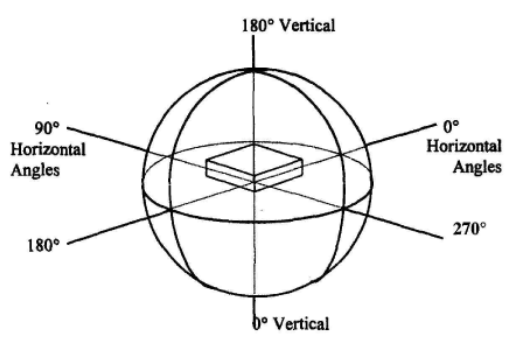
This chapter describes theoretical parts needed to understand how the software was created.

## Relation between an IES file and Unity lights

An IES file (IESNA LM-63 Format) is the most current standard used to describe a light. The most current standard is LM-63-2002, although much photometric data is presented in various older versions of the standard.

An IES file contains data in the ASCII format. A description can be found [here](https://confluence.iir.ntnu.no/download/attachments/67174473/IESNA-LM-63-2-2002-Standard-Free-Download_removed.pdf?version=1&modificationDate=1661962194777&api=v2), but in short this file contains:

* A description of the lamp (manufacturer, catalog, type, wattage...).
* A list of vertical and horizontal angles in which the luminous intensity of the lamp was measured:



* The luminous intensity (in candela) of the lamp for different angles:
  + The luminous intensity for all vertical angles at the first horizontal angle, values separated by spaces and written on one single line.
  + The luminous intensity for all vertical angles at the second horizontal angle, values separated by spaces and written on one single line.
  + ...
  + The luminous intensity for all vertical angles at the last horizontal angle, values separated by spaces and written on one single line.

To import an IES file into Unity, it must be converted to a light cookie. A cookie is a mask placed on a light to create a shadow with a specific shape or color, which changes the appearance and intensity of the light (more information [here](https://docs.unity3d.com/Manual/Cookies.html)). The [Photorealistic lights](https://assetstore.unity.com/packages/tools/utilities/photorealistic-lights-ies-59641) asset is used for the conversion.

The light temperature of Unity lights are set to 6500K (white).

## Luminance computation

The luminance is a photometric measure of the luminous intensity per unit area of light traveling in a given direction (more information on [Wikipedia](https://en.wikipedia.org/wiki/Luminance)). It is expressed in cd/m² (candela per square meter).

Quick note on luminance vs illuminance:

* Illuminance measures the amount of light being projected towards an object by the light (in lux).
* Luminance is the amount of light reflected off the surface being illuminated. It is considered the human perception of brightness.

The luminance can be computed from a color and the camera exposure. In a linear color space, the formula is:

luminance = (*0.0396819152*, *0.458021790*, *0.00609653955) \* pixelColor / exposure*

(see the Luminance() function of [this file](https://github.com/TwoTailsGames/Unity-Built-in-Shaders/blob/master/CGIncludes/UnityCG.cginc)).

In the project, luminance is computed in two different ways:

* The luminance map (see the Luminance map manager part) computes luminance based on what the user sees. The angle of incidence of the light may not be equal to 0.
* The line and grid visualization (see the Light computation Manager part) computes luminance from top, looking at the bottom. The angle of incidence of the light is always equal to 0.

# Source Control

Source Control is done using GitHub. The current repository is available in the [NTNU-IE-IIR team](https://github.com/NTNU-IE-IIR/NORDARK).

[GitHub Actions](https://github.com/features/actions) are used to automate the developer workflow, here the deployment of the project on Windows, MacOS, and Linux. For an overview on Github Actions, you can refer to [this video](https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=video&cd=&cad=rja&uact=8&ved=2ahUKEwjC6-Lj8tL7AhX-QaQEHeyCBgAQtwJ6BAgIEAI&url=https%3A%2F%2Fwww.youtube.com%2Fwatch%3Fv%3DR8_veQiYBjI&usg=AOvVaw06B8n_0gxiO4c28Um2XmVf). The rest of this chapter assumes that you have a basic knowledge of GitHub Actions.

To deploy a Unity application with Github Actions, we have to use the [Unity - Builder](https://github.com/marketplace/actions/unity-builder). A documentation is available [here](https://game.ci/docs/). In the repository, you will find two yaml files in the .github/workflows folder:

* activation.yml: The workflow described by this file is to be manually run once. It provides a Unity license required to run the Unity Builder. You can find more information [here](https://game.ci/docs/github/activation).
* main.yml: This workflow:
  + Is started each time there is a push with a new tag.
  + Build the project for Windows, MacOS, and Linux.
  + Creates a release with the tag and the built versions.
  + On Windows, replaces the NORDARK-WP5\NORDARK-WP5\_Data\Plugins\x86\_64\sqlite3.dll file by the AdditionalResources\sqlite3.dll file. This happens because some computers do not recognize the sqlite3.dll file shipped with the executable.
  + Is freeing disk space at the beginning of the process. This is needed particularly for the MacOS build that needs more space than the total available space of the virtual machine creating the build.

Collaborating on the same project will work for everything except for the Unity scene file (Assets/Scenes/Nordark). Therefore, only one person should modify the Unity scene at a time. The other persons can use prefabs to store game objects.

# Implementation

This chapter describes the implementation of the software and the role of each object in the Unity scene. It is recommended to have the Unity project opened while reading this part.

The current version of Unity used is **Unity 2022.2.0f1.**

## Assets organization

The Assets folder is organized as follow:

* The AdditionnalResources folder contains resources (text, images) that should be shipped with the application (for example IES files).
* The Editor folder contains scripts that should only be executed in edit mode (and not play mode). Currently, it only contains a script executed after each build that ships the resources of the AdditionnalResources folder with the application.
* The HDRPDefaultResources folder contains HDRP settings for the project (see the Global volume and HDRP part).
* The IES folder contains nothing. It is automatically created by the [Photorealistic lights](https://assetstore.unity.com/packages/tools/utilities/photorealistic-lights-ies-59641) asset at each start of the application.
* The Materials folder contains shaders, shader graphs, and materials (derivated from the previous shaders or not).
* The Plugins folder contains external librairies (mainly external assets taken from the [Asset store](https://assetstore.unity.com/)).
* The Prefabs folder contains [prefabs](https://docs.unity3d.com/Manual/Prefabs.html) used in the scene. Prefabs specific to the user interface are in the UI folder.
* The Resources folder contains files that should be accesible after deployment. It contains the [Mapbox access token](https://docs.mapbox.com/help/getting-started/access-tokens/) and the default scene loaded when the application starts.
* The Scenes folder contains the only scene of the project.
* The Scripts folder contains every internal script of the project.
  + The Scripts/Managers folder contains the managers of the application (see the Managers part).
  + The Scripts/Models folder contains several classes describing entities. These classes contain very few methods, their main goal is to provide structures for the other scripts.
  + The Scripts/Objects folder contains scripts attached to prefabs or game objects.
  + The Scripts/UI folder contains scripts attached to UI prefabs or game objects.
  + The Scripts/Utils folder contains scripts that do not fall in any of the previous categories.
* The Textures folder contains textures, sprites, render textures used in the project.
* The Vegetation folder contains assets specific to the Vegetation Studio Pro asset (see the Vegetation Studio Pro part).
* The Volumes folder contains [Volumes](https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@14.0/manual/Volumes.html) settings, for example for the weather.

## Global information on scripts

* To make sure every SerializeField properties of a script are defined, the Assert.IsNotNull(property) function is called in the Awake() function. This helps to identify rapidly if some property is undefined.
* To start the file explorer, the [Unity Standalone File Browser](https://github.com/gkngkc/UnityStandaloneFileBrowser) asset is used, stored in the Assets/Plugins/StandaloneFileBrowser folder.

## Sky (sun and moon)

In the hierarchy, the Sky object controls the sun and the moon of the scene, which are its two children. It has a rotation of 180° along the y-axis, to align the sun/moon position with the Mapbox map (by default, the North/South of the sun/moon are inversed with the North/South of the Mapbox map).

The sun and moon are represented by a directional light whose Affect Physically Based Sky option is checked. The angular diameter, distance, temperature, and intensity parameters are using real-life values. Only the sun create shadows (Shadow Map parameter), because only one directional light can cast shadows at a time.

The Sky object has a SkyManager.cs script attached to it. This script controls the sun/moon position and the moon light intensity:

* The sun/moon position is determined by the [SAMPA algorithm](https://midcdmz.nrel.gov/sampa/). This is a C program, and was converted to C# in the Assets/Scripts/Utils/Sampa folder.
* The moon light intensity changes depending on its phase. The phase is determined by the [MoonPhaseConsole](https://github.com/khalidabuhakmeh/MoonPhaseConsole) algorithm, stored in the Assets/Plugins/MoonPhaseConsole folder.
* The script communicates with the SkyControl.cs script, controlling the corresponding UI element.

## EventSystem

The EventSystem object is an internal Unity object that [process and handle events in a Unity Scene](https://docs.unity3d.com/2019.1/Documentation/ScriptReference/EventSystems.EventSystem.html). It is not directly used but should be present.

## Main camera

The Main Camera object is rendering what we see on the screen.

The CameraMovement.cs script handles the translation/rotation of the camera. The movement speed changes depending on the terrain size.

The SceneCamerasManager.cs script perform several actions related to what the user can see:

* The screen can be split into different views. This is used when comparing different light pole configurations (described below), and is implemented by creating other cameras as children of the Main Camera/AdditionalSceneCameras object. These other cameras are described by the SceneCamera.cs script.
* The script can create a LuminanceMapCamera when the luminance map is activated (see the Luminance map manager part). Also described by the SceneCamera.cs script, this camera sees exactly what the main camera sees but is used to get luminance values (with the GetPixelColorsOfLuminanceMapCameraPointedAroundCursor() function).
* Before each render of the main camera, some light poles will be shown and some will be hidden, depending on which configuration they belong to. This is more detailed in the Light Poles manager part.
* A minimap can be displayed if the user clicks on View/Show minimap. By default, the minimap displays a 2D top view of the scene, but the user can click on it to switch the main camera between a perspective and an orthographic camera (to have a 3D or a 2D view of the scene). In the split screen configuration, the perspective of each camera will be changed. If the perspective is set to orthographic, the size of the cameras can be changed. The camera of the minimap is the Main Camera/Minimap object. It displays what it sees to the Texture/MinimapTexture render texture, used by the UI/MovingWindows/Minimap UI element.
* The script implements the ScreenPointToRay and ScreenToWorldPoint functions. These functions are to be used instead of the Unity [ScreenPointToRay](https://docs.unity3d.com/ScriptReference/Camera.ScreenPointToRay.html) and [ScreenToWorldPoint](https://docs.unity3d.com/ScriptReference/Camera.ScreenToWorldPoint.html) functions, as they will work even with the split screen configuration.

## Global volume and HDRP

This part will explain the GlobalVolume object and how to use the High Definition Render Pipeline (HDRP).

I recommend to read [The definitive guide to lighting in the High Definition Render Pipeline (HDRP)](https://blog.unity.com/games/updated-for-2021-lts-the-definitive-guide-to-lighting-in-the-high-definition-render-pipeline) that explains what HDRP is. This pipeline was chose for this project because it uses real physical units (like Lux, Lumen…) for lights.

The Assets/HDRPDefaultResources folder contain the HDRP settings for the project. The different settings are explained in the guide, but note that you should set the Lit Shader Mode setting to Deferred for the vegetation to work properly (in the Assets/HDRPDefaultResources/HDRenderPipelineAsset file).

These default settings can be overridden in a Unity scene with a global volume. The GlobalVolume object is doing that, by for example setting the sky and the camera exposure.

## Terrain

The Terrain object is responsible for creating and managing the ground of the scene.

Two types of terrain exist:

* The Map object is used to display an accurate terrain based on a location. It is described in the Mapbox section below and its behavior is implemented by the MapManager.cs script.
* The BasicLocation object is used to display a flat 100x100m meters plane. Its behavior is implemented by the BasicLocationManager.cs script and its ground texture can be changed.

The swap between the map or the basic location is handled by the TerrainManager.cs script. This script also contains different functions, for example to convert geographical coordinates into Unity positions. TerrainManager.cs will call the corresponding functions of the map or the basic location depending on the current active type of terrain.

Each type of terrain implements the TerrainTypeManager.cs abstract class, because they share similar functions.

## Mapbox

The Terrrain/Map object is handling the creation of the map and the buildings placed on it. with the AbstractMap.cs script. The map is one of two possible terrains, as described in the Terrain Manager part. The map is generated by the AbstractMap.cs script, which is coming from the [Mapbox Unity SDK](https://www.mapbox.com/unity) and stored in the Assets/Scripts/Utils/Mapbox folder. It has a lot of parameters that can be modified in the Unity inspector, the most important ones being:

* The zoom level of the map.
* The extent of the map (number of tiles generated).
* The tile material (described in the weather and ground texture parts).
* The features (in the Map layers section). A feature is a collection of objects that will be placed on the map. In our case, we use the buildings features.

Other parameters are modified in the MapManager.cs script described just below.

In our case, the AbstractMap is doing four things each time a location is loaded:

* Creating several tiles based on the current location. A tile is a square representing a region of space represented by a [Mesh](https://docs.unity3d.com/ScriptReference/Mesh.html) in Unity.
* Assigning some elevation to each tile. The elevation is fetched from the Mapbox servers under the form of an image.
* Assigning an image to each tile. The image can be for example a satellite picture and is fetched from the Mapbox servers.
* Creating buildings on top of each tile. For each tile, a [GeoJSON](https://en.wikipedia.org/wiki/GeoJSON) file containing features (e.g. roads, buildings) is retrieved from the Mapbox servers. Then, buildings are created using the information stored in the GeoJSON file (mostly position and size). A random texture is assigned to each building.

To interact with the Mapbox map, the MapManager.cs script was created. It is used by different scripts to:

* Change the location of the map. This has an impact on other scripts (for example, the light pole positions need to be updated).
* Convert coordinates between the Unity world and the real world.
* Set the style of the map (e.g. satellite data coming from OpenStreetMap, or a custom texture).
* The script communicates with the MapControl.cs script, controlling the corresponding UI element.

The Map object also contain the VegetationManager.cs and MeshTerrain.cs scripts, which are described in the next section.

It is possible to assign a custom elevation dataset to the ground. This is implemented by the Assets/Scripts/Managers/HeightMapsManager.cs script:

* First, the script lets the user import GeoTIFF files. GeoTIFF is format extension for storing georeference and geocoding information in a TIFF 6.0 compliant raster file by tying a raster image to a known model space or map projection.
* To process the GeoTIFF files, the LibTiff library from the Assets/Plugins/LibTiff folder is used. [This library](https://bitmiracle.com/libtiff/) is used to read TIFF images (and not GeoTIFF), so I used [this guide](http://build-failed.blogspot.com/2014/12/processing-geotiff-files-in-net-without.html) to get the "geo" information out of the tiff file.
* The goal of the process is to create the same type of data that Mapbox uses to represent elevation. This type of data is described [here](https://docs.mapbox.com/data/tilesets/guides/access-elevation-data/). For each tile, there is an image that contains elevation data in its red, green, and blue pixel components. The following equation is used to get the elevation from a pixel color:

elevation = -10000 + (({R} \* 256 \* 256 + {G} \* 256 + {B}) \* 0.1)

Here is the process in details:

* + First, we get the geographical coordinates of the GeoTIFF file by reading specific "tags" of the image. Some pixels of the image have a special value used to represent "no data", so we also get that special value.
  + Then, for each tile of the current map, we read the pixels of the GeoTIFF file corresponding to that particular tile. Each pixel value corresponds to an elevation in meters. We create the Mapbox elevation image of these elevation values using the reciprocal of the formula above.
  + Mapbox elevation images always have a size of 256x256 pixels, so we resize our created images to fit these dimensions.
  + Finally, the created images are saved to a persistent data directory.
* We now have to tell Mapbox to use these new elevation images. Elevation data is fetched in the Assets/Scripts/Utils/Mapbox/Unity/MeshGeneration/Factories/TerrainDataFetcher.cs script, so I created the Assets/Scripts/Utils/MapboxElevationCustomization/CustomTerrainDataFetcher.cs script that is a copy of the TerrainDataFetcher.cs script with some modifications to use the new elevation images. To tell Mapbox to use the new CustomTerrainDataFetcher.cs script, the Assets/Scripts/Utils/Mapbox/Unity/MeshGeneration/Factories/TerrainFactoryBase.cs script has been changed a bit.
* You may notice that the Mapbox files are stored in the Scripts folder, and not in the Plugins folder. This is because Unity compiles scripts of the Plugins folder before scripts of the Scripts folder. As we changed the behavior of the TerrainDataFetcher.cs script to use the CustomTerrainDataFetcher.cs script, TerrainDataFetcher.cs cannot be compiled before CustomTerrainDataFetcher.cs. Therefore, it was necessary to put the Mapbox files in the Scripts folder.

## Vegetation Studio Pro

This part explains the VegetationStudioPro object and how to use this asset.

The [Vegetation Studio Pro](https://assetstore.unity.com/packages/tools/terrain/vegetation-studio-pro-131835) asset creates vegetation randomly on user-defined areas. Each area is using a certain biome, which is a collection of trees/grass/bushes/rocks. You can find a detailed guide [here](https://www.awesometech.no/index.php/setup-guide/).

As the time of writing, two biomes have been defined: AlesundForest and UppsalaForest. They are stored in the Assets/Vegetation/Biomes folder. To create a new biome, right click on this folder, expand Create, find Awesome Technologies in the bottom, VegetationPackagePro, No Texture Biome.

The object controlling the vegetation is VegetationSystemPro (a child of the VegetationStudioPro object). Click on this object, find the Biomes tab on the inspector, and you can add a new biome by dragging your newly created biome in the Add Vegetation Package parameter. Then, modify the Select Biome parameter (use Biome 3 for example). Then, go to the Edit Biomes tab. Select your biome with the Selected vegetation package parameter. You can now add trees/grass/bushes/rocks to this new biome by dragging these objects in the Add Vegetation Items tab.

When you select an object (in the Add Vegetation Items tab), you can modify some parameters related to the generation of this object, the most important ones being:

* Seed: you should assign a value different from all the one of the other objects of the same biome.
* Density: this controls the number of times the object will be spawned.
* Enable billboards: disable this checkbox if there is a bug with how the tree looks. This is generally the case for trees not specifically designed to be used by the Vegetation Studio Pro asset.

If the leaves of a tree don’t appear, it may be because the tree is using one game object for the trunk and another for the leaves. The Vegetation Studio Pro asset doesn’t support such objects. To adapt these trees:

* Drag the tree to the Unity scene and unpack it (right click, Prefab, Unpack completely).
* Create an empty game object with the same name, set its position to be the origin. Add the trunk and the leaves game objects as direct children of this game object.
* Add the Mesh Combiner script to the parent game object. Check Create Multi-Materials Mesh and Destroy Combined Children.
* Click on Combine Meshes and Save Combined Mesh. Remove the Mesh Combiner script. You can now create a prefab from this game object that will be compatible with the Vegetation Studio Pro asset.

If you look at the Map object, you can notice the BiomeAreasManager.cs and MeshTerrain.cs scripts. The MeshTerrain.cs script is coming from the Vegetation Studio Pro asset and is used to link the map with the vegetation. The BiomeAreasManager.cs script is a custom script that:

* Can create, modify, and delete biome areas. A biome area is a user-defined geographical area on which vegetation from a certain biome should be spawned. A biome area is defined by:
  + A biome.
  + Several nodes. A node is geographical point, and the vegetation should be spawned in the area within all nodes. The user can add or delete nodes.
* Can generate biome areas, meaning actually spawning the vegetation on the map. This has to be done each time the location is changed.
* Can change the density (see above) of each biome.
* Is only working when the current location is of map terrain type. Therefore, the corresponding UI element is disabled when the current location terrain type is not a map.
* The script communicates with the BiomeAreasControl.cs script, controlling the corresponding UI element.
* The script implements the ObjectsManager abstract class. This class describes entities associated with locations.

Vegetation is rendered on the areas that the cameras can see. As such, the main camera, the street view camera (detailed in the street view manager part), the cameras detailed in the cameras manager part, and the cameras used for luminance computation are added to the Cameras section of the VegetationSystemPro object.

During development, when you don't need vegetation, I recommend disabling the VegetationStudioPro object. This will save you a few seconds each time you compile scripts and run the application.

## UI

The UI object contains the user interface. Usually, scripts related to the UI are called somethingControl.cs. Inside such scripts, the initialization is done in the Awake() function, and the listener are defined in the Start() function. Some UI elements use the Horizontal Layout Group or Vertical Layout Group components, which automatically set the position of child UI elements.

The UI object contains three scripts:

* ShorcutsControl.cs, that listens to keyboard events.
* UIController.cs, that initializes the UI.
* DialogControl.cs, that can be used to create an informative dialog or a confirmation dialog.

From top to bottom of the screen, there is:

* The menu bar (MenuBar), composed of buttons that can open panels. The MenuBarPanelsControl.cs script is handling the user interaction with the buttons, and there is one script per panel for specific actions related to the panel. In the hierarchy, the MenuBar object is below the tool bar and the tabs, because the panels must appear on top of these objects. To add a new button, add a button in the Bar object and a panel in the Panels object.
* The tool bar (ToolBar), composed of four buttons to save/save as a scene, change between the 2D and 3D view, use a measure line, and other buttons to switch between the different tabs. The ToolBarTabsControl.cs script, on the Tabs object, is handling the user interaction. To add a new tab, add a button to the Tabs object and see the next item.
* The tabs (Tabs), present on the left of the screen. Each tab is divided into sections, each section having a script attached to it. To add a new tab, add a child to the Tabs object and add a script that implements the TabControl class to it.

Other UI elements are:

* Under the MainScreenPanels object:
  + A location undefined panel (LocationUndefined) that is shown when there is no active location.
  + A loading screen (Loading) that is shown when the application starts until the map is loaded.
* Under the MovingWindows object:
  + A camera preview (CameraPreview) that is displayed when the camera tab is active. This window can be moved by the user thanks to the EventTrigger component and the Drag.cs script.
  + A graph (Graph) that is displayed when it is enabled in the light computation tab. The graph is controlled by the GraphControl.cs script, and the window can also be moved. The graph can be a line graph or a bar chart, and can display multiple sets of points at the same time.
  + A heatmap (Heatmap) that is displayed when it is enabled in the light computation tab. The heatmap is controlled by the HeatmapControl.cs script, and the window can also be moved. The actual heatmap (not the heatmap window) is controlled by the Heatmap.cs script. This scripts implements the MaskableGraphic class, which allows to create custom meshes and thus perform a [triangulated irregular network](https://en.wikipedia.org/wiki/Triangulated_irregular_network) interpolation.
  + A minimap (Minimap), take a look at the main camera part to have more information on it.
* Under the Legends object:
  + A legend for the light luminance (LuminanceMapLegend) that is displayed when the display luminance map toggle is on.
  + A legend for the visualization tab (VisualizationLegend) that is displayed when the data visualization tab is on.
* Under the Windows object:
  + A window (NewLocation) to let the user create a new location.
  + A window (Controls) to show the user how to move in the scene.
  + A window (GroundTextures) displaying where ground textures are created. This is a modal window, which means the user can not interact with the rest of the UI when this window is active.
  + A window (AddVariable) to let the user add a variable to an existing dataset in the data visualization tab.
  + A window (MeasureLineParameters) to help the user when visualizing luminance along a line in the light computation tab.
* A tooltip (Tooltip) which is shown when the user hovers on a visualization feature or a point in the graph.

## Managers

The Managers object contain empty game objects with scripts attached to it. They handle everything but the UI. Some managers are grouped based on the features they provide.

### IES Manager

The IESManager.cs script is handling everything related to IES files:

* Information on IES files can be found in the theory part of this documentation.
* Default IES files are stored in the Assets/AdditionnalResources/IES folder. During deployment, these files are shipped with the application and so will be accessible to the user. Take a look at the Game manager part to see how it is done.
* An IES light is represented by the IESLight class.
* The user can upload IES files.
* To get the light intensity value from an IES file, the Assets/Scripts/Utils/IESReader.cs script is used. This is a script adapted from the built-in Unity solution for handling IES file. The whole Unity feature for reading IES files is not used because it is only available in the Unity editor, and not after the deployment process.
* To convert an IES file to a cookie, the [Photorealistic lights (IES)](https://assetstore.unity.com/packages/tools/utilities/photorealistic-lights-ies-59641) asset is used, stored in the Assets/Plugins/IES folder.

### Light Poles manager

The LightPolesManager.cs script is responsible for creating, modifying, and deleting light poles:

* A light pole is represented by the LightPole class. The Unity object of a light pole is represented by the LightPrefab class.
* The script implements the ObjectsManager abstract class. This class describes entities associated with locations.
* One or several light poles can be selected. They can then be modified via the user interface.
* The LightPolesSelectionManager.cs script lets the user define a rectangle to select some light poles.
  + If the current view is in 2D, all light poles within the rectangle are selected.
  + If the current view is in 3D, rays are sent in different positions within the rectangle. If a ray intersects a light pole, the light pole becomes selected.
* The script communicates with the LightPolesControl.cs script, controlling the corresponding UI element.
* The script uses the Assets/Materials/Highlighted material when requested to highlight the light poles.
* The user can add light poles to the scene by clicking on Insert, Lights. The user must provides a GeoJSON file with a collection of points, and the coordinates of this file must use the EPSG:4326 coordinate system (longitude from -180° to 180° / latitude from -90° to 90°). Converting a GeoJSON file into features is done using the GeoJSONParser class, defined in the Assets/Scripts/Utils folder. To serialize / deserialize a GeoJSON file, the [GeoJSON.NET](https://github.com/GeoJSON-Net/GeoJSON.Net) library is used. It is stored in the Assets/Plugins/GeoJson.NET folder.
* Each light pole located outside of the current terrain is disabled to save performance.
* Each light has a configuration index assigned to it. This corresponds to a configuration a light pole belongs to. For example, the first configuration is described by index 0, the second configuration by index 1, and so on. The number and the definition of configurations can be done in the Light Computation tab. The configuration index is used by cameras. For example, when computing luminance results for the first configuration, all the lights with the configuration index 0 will be shown and all the other lights will be hidden.
* Each light pole can be assigned to one or several groups. This is implemented by the LightPolesGroupsManager.cs script. A group is just identified by its name, and can be used to apply modifications to multiple light poles at the same time without having to manually select them beforehand.

### Cameras manager

The CamerasManager.cs script is responsible for creating, modifying, and deleting cameras:

* A camera is represented by the CameraNode class. The Unity object of a camera is represented by the CameraPrefab class.
* The script implements the ObjectsManager abstract class. This class describes entities associated with locations.
* The script communicates with the CameraControl.cs and CameraParametersControl.cs scripts, controlling the corresponding UI elements.
* The cameras manager object contains a global volume that sets the exposure and the depth of field modes to physical camera. This object uses the 7:Camera Unity layer, so it only applies to cameras using the same layer, here the cameras managed by CameraManager.cs but not the main camera. Note that this volume has a priority of 1, so it overrides the volume of the GlobalVolume object (which has a priority of 0).

### Vegetation objects manager

The VegetationObjectsManager.cs script is responsible for creating, modifying, and deleting vegetation objects:

* A vegetation object is any kind of 3D object like a tree or some grass. It is represented by the VegetationObject class, and the Unity object of a vegetation object is represented by the VegetationObjectPrefab class.
* The script implements the ObjectsManager abstract class. This class describes entities associated with locations.
* The script communicates with the VegetationObjectsControl.cs script, controlling the corresponding UI elements.
* One or several vegetation object can be selected. They can then be modified via the user interface.

### Locations manager

The LocationManager.cs script is responsible for creating and deleting locations.

* A location is represented by the Location class.
* The script communicates with the LocationControl.cs script, controlling the corresponding UI element.

### Ground Textures manager

The GroundTexturesManager.cs script is responsible for defining custom textures that should be applied to the ground.

* A ground texture is represented by the GroundTexture class. This corresponds to a texture and a geographical area where the texture should be applied. Ground textures are grouped into GroundTextureCollection, each GroundTextureCollection containing ground textures imported by the user with the same file.
* The script implements the ObjectsManager abstract class. This class describes entities associated with locations.
* As described in the Mapbox section, the map is divided into tiles, which are square represented by Unity meshes. Therefore, defining a new texture to a certain area requires to modify one or more tiles. The general idea is to create a "masks" for each tile and each textures, where each mask is a black and white image representing where to place a certain texture on a certain tile. The mask creation is done in the GroundTextureMasksManager.cs script.
* Here is how new ground textures are created:
  + First, the user has to provide a GeoJSON file containing polygons representing ground textures. The coordinate system (or CRS) of the file must be EPSG:4326, otherwise the import is not done.
  + Then, each polygon is converted into a GroundTexture object. The obtained set of GroundTexture objects is stored into a new GroundTextureCollection object. We filter the obtained set of GroundTexture objects by only keeping ground textures located on the current map.
  + Then, for each type of texture:
    - First, Unity meshes are created based on the coordinates defined in all ground texture objects that describe the current texture. To convert a set of coordinates into triangles (needed for the Unity mesh), the Earcut library is used. It is part of the Mapbox SDK and is defined in the Assets/Scripts/Utils/Mapbox/Unity/MeshGeneration/Modifiers/MeshModifiers/Earcut.cs file. The mesh is created a few meters below the map. Two types of meshes are created:
      * Some white meshes, that define areas where to place the ground textures.
      * Some black meshes, that define areas where not to place the ground textures. This comes from the [GeoJSON specifications](https://www.rfc-editor.org/rfc/rfc7946#section-3.1.6): polygons consist of an array of linear ring coordinate arrays, where the first one is the exterior ring and the following one are interior rings, i.e. holes in within the exterior ring.
    - Then, the goal is to determine whether the meshes are below certain tiles. For this, an orthographic camera (Assets/Prefabs/MaskCamera game object) will move to the center of each tile, looking down. The viewing size of the camera is the same as the size of each tile, so the camera will exactly see what is placed directly below the tile. What the camera sees is rendered to a [RenderTexture](https://docs.unity3d.com/Manual/class-RenderTexture.html), then copied to a [Texture2D](https://docs.unity3d.com/ScriptReference/Texture2D.html), which will be passed to the tile material. For example, if no part of the meshes is below a tile, the texture will be completely black (the background color). If part of the meshes is below the tile, part of the texture will be white (the color of the mesh) and part of the texture will be black. Thus, this step creates a mask for each tile, with the black color meaning no ground texture and the white color meaning ground texture. If you look at the code, you'll notice that this process is done in a [coroutine](https://docs.unity3d.com/Manual/Coroutines.html). The yield return null; command is used at each tile to skip a frame, which is needed for the camera to render to the texture.
    - The tile material is located in the Assets/Materials/Tile folder. This material was generated by the Assets/Materials/Tile/Tile shader graph. [Shader graphs](https://docs.unity3d.com/Packages/com.unity.shadergraph@13.1/manual/index.html) are the usual way to create shaders with Unity HDRP. If you double click on the Tile shader graph, you'll see:
      * A list of parameters on the left, which are the inputs of the shader, divided in sections.
        + The Base Ground section contains the texture applied by default to the entire map.

Diffuse Map: the base color of the object. For the map, this is generally a satellite image specific to each tile, so it's modified by script.

Normal Map: no texture is defined here, the default normals are used.

Mask Map: defines several properties, described [here](https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@7.1/manual/Mask-Map-and-Detail-Map.html#mask-map). We use the same transparent green texture defined in the Assets/Textures/TileDefaultMaskMap file for each tile.

Emission Map and Emission color: no texture is defined here.

* + - * + The Snow and Wetness sections contains parameters defined in the weather section of this documentation.
        + The Groundi (i going from 0 to 7) sections contain the parameters for each type of ground texture.

Diffusei, NormalMapi, HeightMapi, MaskMapi: the textures for one type of ground texture. The height map is a grayscale texture that [shifts the areas of the visible surface texture around](https://docs.unity3d.com/Manual/StandardShaderMaterialParameterHeightMap.html).

Tilingi: 2-dimensionnal vector modifying the UV coordinates. The greater the tiling, the smaller the texture appears.

Maski: black and white mask defining where the ground texture should be applied. If a pixel is white, the ground texture should be applied. If it's black, the default texture (or another ground texture) should be applied. The texture previously created by the orthographic camera is passed to this parameter, so this texture is modified by script.

Texturei: float indicating which type of texture is used (for example grass, wetland, road). A value of -1 means no texture, and a positive value corresponds to the index of a type of texture in the textureNames variable defined in the GroundTexturesManager.cs script.

* + - * The actual shader graph, divided in 4 columns.
        + The first column contains all the inputs of the shader. The 8Grounds node is a [sub graph](https://docs.unity3d.com/Packages/com.unity.shadergraph@7.1/manual/Sub-graph.html), defined in the Assets/Materials/Tile/Ground folder. It is responsible for merging eight types of ground textures and uses the 4Grounds sub graph defined in the same folder. The 4Grounds sub graph uses the 2Grounds sub graph, and the 2Grounds sub graph uses the Ground sub graph.
        + The second column merges the different ground textures with the default texture. It uses the ground texture mask to determine which texture (default or ground texture) should be applied.
        + The third column modifies some of the outputs of column 2 based on the weather conditions. It is described in the weather part of this tutorial.
        + The fourth column contains the vertex and fragment nodes, the output of the shader.
    - Finally, the initial white meshes are deleted.
  + This description was actually a simplification of what is actually done. Here are some more advanced features that are implemented:
    - The process of creating masks takes a lot of time. Therefore, it is only done once and the masks are saved in an application folder when created. Each ground texture collection has a unique [Globally unique identifier](https://fr.wikipedia.org/wiki/Globally_unique_identifier) which is used to name the folder containing the png files for one ground texture.
    - The created masks use the MaskMesh Unity layer. If you look at the Main Camera of the scene, you'll see its Culling Mask parameter does not include the MaskMesh layer, so that the main camera does not render the masks.
    - When creating a mask for one tile and one type of texture, if a mask already exists, then the new mask must be combined with the already existing mask. This could be done by reading each pixel of the two masks one by one, but this would take a lot of time. Instead, the CombineMasks [compute shader](https://docs.unity3d.com/Manual/class-ComputeShader.html) (defined in Assets/Materials/MaskMesh folder) is used to perform this process on the GPU, making it faster.
    - A maximum of 8 different ground textures can be applied to one tile. This restriction comes from the fact that no more than 64 textures can be assigned to one shader, and each ground texture consists of multiple textures (diffuse, normal map...).
    - If a mask is completely black, then no ground texture should be applied. This is detected by the DetectBlackMask compute shader (defined in Assets/Materials/MaskMesh folder).

It is possible to add ground textures to the scene by clicking on Insert, Ground Textures. The user must provides a GeoJSON file with a collection of polygons, each polygon having a property (integer) that indicates which texture to use.

Pre-computed masks are stored in the Assets/AdditionnalResources/ground-textures folder. During deployment, these files are shipped with the application and so will be accessible to the user. Take a look at the Game manager part to see how it is done.

### Data visualization manager

The DataVisualizationManager.cs script provides a visualization of 2D data.

* A dataset is represented by the Dataset class. A dataset is a collection of variables assigned to geographical areas. For example, the walkability dataset contains the speedLimit and population variables, so each road of this dataset has 2 values: one representing the speed limit, and the other representing the population. The user can apply a weight to each variable, and the resulting value is a linear combination of all variables with their corresponding weights.
* The script implements the ObjectsManager abstract class. This class describes entities associated with locations.
* The script communicates with the DataVisualizationControl.cs script, controlling the corresponding UI element.
* A dataset contains a collection of VisualizationFeature objects. A VisualizationFeature object is an instance of the VisualizationFeature prefab (defined in the Assets/Prefabs folder) with a VisualizationFeature.cs script (defined in the Assets/Scripts/Prefabs folder) attached to it. A visualization feature is a 3D object representing a value:
  + If the input data is a set of lines, each visualization feature will be a 3D line. The color of each line represents its value.
  + If the input data is a set of points, each visualization feature will be a 3D vertical bar. The color and the height of each bar represents its value. The size and height of each bar changes depending on the terrain size.
  + If the input data is a set of polygons, the [centroid](https://en.wikipedia.org/wiki/Centroid) of each polygon is computed, and the same visualization as for the points is used.
* A tooltip (UI/Tooltip in the hierarchy) is displayed when hovering a visualization feature. It displays what indicator is currently selected and the associated value of that specific visualization feature.
* The user can add a variable to an existing dataset by providing a GeoJSON file. This file must contain a set of points, and a value will be assigned to each feature of the existing dataset by looking at the nearest feature of each point. A use case of this feature is to compute luminance results in the Light Computation tab, export the results to a GeoJSON file, and upload this file in the add variable window.

### Game manager

The GameManager.cs script is responsible for initializing the software. Basically, everything should be started from here (except internal initialization steps like creating lists, setting up listeners on UI elements..):

* It request for the UI to set up.
* It loads the default scene after the map is initialized.

Basic data (for example sample IES lights) can be saved in the Assets/AdditionalResources folder to be made accessible to the user. This is done with the Editor/AdditionalBuildCommands.cs script. This script is only executed after each deployment, and its goal is to ship the content of the Assets/AdditionalResources folder with the application. If it is the first time that the user starts the application (with this specific version), the game manager will tell some managers to load this data. To determine if this is the first time the user starts the application, the [PlayerPrefs](https://docs.unity3d.com/ScriptReference/PlayerPrefs.html) are used.

### Scene manager

The SceneManager.cs script handles saving and loading scenes:

* A scene is described by a GeoJSON file with the .nordark extension that contains information on light poles, cameras, locations, biome areas, ground textures, datasets, and vegetation objects:
  + A light pole is represented by a Point, and contains the following properties:
    - type: a string set to light.
    - location: a string describing the name of the light pole's location.
    - name: a string describing the light pole.
    - eulerAngles: a list of 3 floats describing the rotation of the light.
    - IESFileName: a string describing the name of the corresponding IES file.
    - prefabName: a string describing the name of the corresponding Unity prefab name.
    - height: a float describing the height of the light pole.
    - groups: a list of string describing the group names this light pole belongs to.
  + A camera is represented by a Point, and contains the following properties:
    - type: a string set to camera.
    - location: a string describing the name of the camera's location.
    - name: a string describing the camera.
    - eulerAngles: a list of 3 floats describing the rotation of the camera.
    - parameters: an object defining the camera physical parameters and containing the following properties:
      * SensorSize: a list of 2 floats describing the size of the camera sensor in millimeters.
      * ISO: an integer describing the light sensitivity of the sensor.
      * ShutterSpeed: a float describing the amount of time the camera sensor is capturing lights in seconds.
      * FocalLength: a float describing the simulated distance between the lens and the sensor of the camera.
      * Aperture: a float describing the f-stop (f-number) of the lens.
      * Shift: a list of 2 floats describing the offset from the camera sensor, measured as a multiple of the sensor size.
  + A biome area is represented by a Polygon, and contains the following properties:
    - type: a string set to biomeArea.
    - location: a string describing the name of the biome area's location.
    - name: a string describing the biome area.
    - biome: a string describing the biome name of this biome area.
  + A location is represented by a Point, and contains the following properties:
    - type: a string set to location.
    - name: a string describing the location.
    - cameraCoordinates: a list of 3 floats describing the coordinates of the main camera for this location.
    - cameraAngles: a list of 3 floats describing the rotation of the main camera for this location.
  + A ground texture collection is represented by a Point (but this point is not used), and contains the following properties:
    - type: a string set to groundTexture.
    - location: a string describing the name of the ground texture's location.
    - content: a string containing the feature collection provided by the user.
    - id: the unique id of the ground texture collection.
  + A dataset is represented by a Point (but this point is not used), and contains the following properties:
    - type: a string set to dataset.
    - location: a string describing the name of the dataset's location.
    - content: a string containing the feature collection provided by the user.
    - name: the name of the dataset.
  + A vegetation object is represented by a Point, and contains the following properties:
    - type: a string set to vegetationObject.
    - location: a string describing the name of the vegetation object's location.
    - eulerAngles: a list of 3 floats describing the rotation of the object.
    - prefabName: a string describing the name of the corresponding Unity prefab name.
* A default scene, loaded when the application is started, is stored in the Assets/Resource/defaultScene.json file. This scene contain data for the study areas of Alesund and Uppsala.

### Street view manager

The StreetViewManager.cs script handles the street view.

### Luminance map manager

The LuminanceMapManager.cs script computes and provides a luminance map. Information on luminance can be found in the theory part of this documentation. The luminance map converts what the main camera sees into colors representing different luminance values.

* The LuminanceMapManager.cs script communicates with the LuminanceMapControl.cs script, controlling the corresponding UI element.
* The LuminanceMapManager.cs script is enabling/disabling the legend and the LuminanceMapPassAndVolume object:
  + The legend is described by the LuminanceMapLegend.cs script. It can set the value corresponding to each color. The minimal and maximal values of the legend can be specified by the user, as well as the scale (linear or logarithmic).
  + The LuminanceMapPassAndVolume object contains a [Custom Pass Volume](https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@14.0/manual/Custom-Pass.html) that controls how Unity renders the objects in a scene.
    - Unity executes a Custom Pass at a certain point during the HDRP render loop. This custom pass can be injected anywhere in the render loop, before post process in our case.
    - The custom pass is described by a material, itself generated by a shader. Here, the Assets/Materials/Light Computation/FullScreen\_LuminanceMapCustomPass material is used, generated by the Assets/Materials/Light Computation/LuminanceMapCustomPass shader.
    - This shader is a fragment shader. We convert the pixel color into a luminance, and then convert the luminance to color. The relevant lines of code are between lines 5 and 7, and between lines 24 and 39. The legend values and colors are passed to the shader.
  + The LuminanceMapPassAndVolume also contains a [Volume](https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@16.0/manual/Volumes.html) to disable after effects (tonemapping and bloom) because they alter the luminance computation.
  + Note that the LuminanceMapPassAndVolume object uses the LuminanceMap layer. In doing so, this object has only an impact on cameras whose Volume Mask parameter includes this layer. For example, the main camera is affected by this object, but not the cameras used to compute luminance (described in the next section).
* When the luminance map is activated, the UI displays the luminance value currently pointed by the cursor:
  + As the luminance map displays one color for a range of luminance values, we cannot directly use what the main camera sees to determine the luminance value pointed by the cursor (there would be a loss of precision).
  + Instead, we use a second camera with the same parameters as the main camera except for the Volume Mask parameter which does not include the LuminanceMap layer. This camera is created in the SceneCamerasManager.cs script (CreateOrDeleteLuminanceMapCamera() function).
  + The Volume layer of the previously created camera includes the LuminanceInLuminanceMap layer, to make it affected by the LuminancePassAndVolume object (child of the LuminanceMapManager object). This object is similar to the LuminanceMapPassAndVolume object, except that the custom pass volume only affects one camera, and it uses the Assets/Materials/Light Computation/FullScreen\_LuminanceCustomPass material generated by the Assets/Materials/Light Computation/LuminanceCustomPass shader. This material/shader is described in details in the Light computation manager section, but in short it encodes the luminance value into the red, green, and blue components of each pixel.
  + To determine the luminance value pointed by the cursor, we simply read the pixel color of the previous camera at the cursor's position, and convert this color into a luminance value. To have smoother values, we don't actually only read one pixel, but also neighboring pixels and an average of all luminance values is made.

### Light computation manager

The LightComputationManager.cs script computes and provides visualizations of luminance.

* The LightComputationManager.cs script contains generic functions used by the different visualization methods. The most important function computes and returns luminance results on different positions:
  + The general idea is to:
    - Create an orthographic camera that looks down along the different positions.
    - For each position, each pixel of what the cameras sees is converted to a luminance. This creates a texture where each pixel contains a luminance value.
    - On each previously created texture, the average luminance is computed. We get one luminance value for each position of the camera, so we have an array of luminance values associated to an array of positions.
    - These luminance and position arrays are passed to the visualization methods.
  + First, the LuminancePass object (under the LightComputationManager object from the hierarchy) is activated. This object contains a custom pass volume (like the LuminanceMapPass object). This pass is described by the Assets/Materials/Light Computation/FullScreen\_LuminanceCustomPass material generated by the Assets/Materials/Light Computation/LuminanceCustomPass shader. This shader converts the pixel color into a luminance which is returned by the shader. A shader returns a color, so four values between 0 and 1, but here we want to return the luminance which is a value between 0 and around 15000 (the luminance on a bright day). One way would be to divide the luminance by 15000 (which then would be a value between 0 and 1), but this creates precision issues for small luminances between 0 and 1. Instead, the luminance is decomposed into 3 numbers: one number representing the luminance with a precision of 100, one number representing the luminance with a precision of 1, and one number representing the luminance with a precision of 0.01.
  + In order for the LuminancePass object to be active, a frame needs to be skipped. This is one of the reasons this function is a coroutine.
  + Then, an orthographic camera is created from the Assets/Prefabs/LuminanceCamera object. The camera will not render the lines used by the visualization methods (its Culling Mask parameter does not include the Line layer) and is affected by the LuminanceCameraVolume object (under the LightComputationManager object from the hierarchy). The LuminanceCameraVolume object contains a global volume with two overrides: it disables the [tonemapping](https://en.wikipedia.org/wiki/Tone_mapping) and bloom (enable / disable it under the GlobalVolume object and look at the sun to see its effect) after effects because they interferes with the luminance computation. Indeed, the LuminancePass custom pass volume is executed before post process in the render loop. Therefore, any after effect will change the color of the pixels, but in this section we encoded the luminance into the pixel color, so we don't want this to happen. The luminance camera is controlled by the LuminanceCamera.cs script. This script handles the creation and deletion of the camera, and it also only displays light poles that belong to the same configuration (as described in the LightConfigurationsManager below).
  + For each light configuration (which are described below), the camera will go to different positions. For each position, what the camera sees is rendered to a texture. A frame needs to be skipped at each position for the camera to render. As it was described before, the texture contains the luminance values of each pixel, but we only want one luminance value. Therefore, the average luminance of the texture is computed. This is done by the Assets/Materials/Light Computation/LuminanceSum compute shader, because using the GPU for such tasks is faster than using the CPU.
  + We obtain an array of luminances, that can be passed to the visualization methods.
* Two visualizations methods have been developed. Both methods implement the IComputationObject.cs interface:
  + A luminance visualization along lines with the ComputationLine object.
    - The lines are created by selecting two or more positions. It's a LineRenderer Unity object.
      * The line is placed slightly above the ground
      * When creating the lines, the left click is used to start a new line and the right click is used to end adding the line. A tooltip indicates the current length of the line and the angle between the line and the North.
      * Once the line are created, a mesh collider is added to them. This is used to detect when the mouse is hovering the lines. In that situation, a tooltip indicates the distance of the point hovered by the cursor to the origin of the lines.
    - Once the lines are created, the luminance values are computed along them by the LightComputationManager.cs script.
    - The computed luminances for each position are given to the Graph object of the UI through the GraphControl.cs script, that provides a visualization of them.
    - When the cursor is hovering the lines, a red line on the graph indicates to which point it corresponds.
    - The computed results can be exported to a GeoJSON file.
    - Real life measurements can be imported through a GeoJSON file. When the import is done, the luminance will be computed on the same locations as the real life measurements and the graph will display both.
    - The minimal and maximal values of the graph can be specified by the user. If not, the graph will scale according to the current set of values.
    - It is also possible to easily create a measure line starting from a light pole. When the user clicks on the Create measure line around light pole button, a window opens and lets the user select a light pole, the line length, the line angle with the North, the distance between measure points, and a new measure line with these parameters will be created. This behavior is implemented by the MeasureLineParametersWindow.cs script.
  + A heatmap visualization on a rectangle with the ComputationRectangle object.
    - The rectangle is defined by selecting one position and dragging the mouse and consists of four lines. It's a LineRenderer Unity object.
      * The lines are placed slightly above the ground.
      * When creating the rectangle, the left click is used to set the rectangle center and the right click to finish editing the rectangle. A tooltip indicates the current width and length of the rectangle.
    - Once the rectangle is created, the luminance values are computed on different positions of the rectangle by the LightComputationManager.cs script.
    - The computed luminances for each position are given to the Heatmap object of the UI through the HeatmapControl.cs script, that provides a visualization of them.
    - When the cursor is hovering the heatmap, a red line in the scene indicates to which location it corresponds.
    - The computed results can be exported to a GeoJSON file.
    - The minimal and maximal values of the heatmap can be specified by the user. If not, the heatmap will scale according to the current set of values.
    - Values can be displayed on the heatmap. Each value is a HeatmapPoint object.

### Light configurations manager

The LightConfigurationsManager.cs script lets the user define several configurations of light poles.

* A configuration is an arrangement of light poles.
* The LightConfigurationsManager.cs script communicates with the ConfigurationsControl.cs script, controlling the corresponding UI element.
* The current scene is considered to be the default configuration. The user can increase the number of configurations, up to 4 at the time of writing. A configuration is a scene file, as described in the Scene manager part with the .nordark extension.
* When importing a configuration, only the light poles will be taken into account (and not the cameras for example).

### Weather manager

The WeatherManager.cs script handles the weather system: snow, rain, clouds, fog:

* Each type of weather is described by a game object child of WeatherManager:
  + The Fog object contains a Unity volume with a [*Fog override*](https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@15.0/manual/Override-Fog.html#customizing-global-fog). Its weight is changed by script.
  + The Clouds object contains several children, each one containing a Unity volume with a [*Volumetrics Clouds override*](https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@15.0/manual/Override-Volumetric-Clouds.html#shape). The Cloud Preset parameter is different for each of these objects, and they get enabled/disabled by script. The volumes are stored in the Assets/VFX folder.
  + The Rain/Snow objects contains a particle system that spawn rain/snow particles. The particles are adapted from the [10 Effects - Colorful FX Package](https://assetstore.unity.com/packages/vfx/particles/10-effects-colorful-fx-package-184732) asset, and only spawn around the main camera with the WeatherParticlesPrefab.cs script. The max number of particles and the spawn rate can be changed by script.
* The terrain is also modified depending on the precipitations (rain/snow). This is done using shaders, as described in the Ground Textures manager section of this documentation, and more precisely with the Assets/Materials/Tile/Weather/Weather sub graph, composed of two sub graphs:
  + Assets/Materials/Tile/Weather/Wet, that creates the wet aspect on the ground. It was adapted from [this video](https://www.youtube.com/watch?v=jZbG9zFvSJE&feature=emb_logo), and has one important parameter: \_Wetness, float between 0 and 1, that determines the amount of water on the ground. In a script, this parameter can be set for all instances of the material with the Shader.SetGlobalFloat() function.
  + Assets/Materials/Tile/Weather/Snow, that add snow to the ground. It was adapted from [this video](https://www.youtube.com/watch?v=IC9g5hlfV6o), and has one important parameter: \_Snow\_Opacity, float between 0 and 1, that determines the amount of snow on the ground.

### Measure manager

The MeasureManager.cs script lets the user define a multiline in the scene that can be used to measure distances. Its behavior is very similar to the one of the ComputationLine object.

## Splash screen

The splash screen can be changed in the player settings, that can be opened by clicking on Edit, Project Settings…, Player tab, and Splash image section. The splash images are stored in the Assets/Textures folder.

## Assets

A list of all assets can be found in [Development environment](file:////display/NORDARK/Development+environment).