You can find the last version of this guide at this link.

Overworld 2D Generator

By Andrea Novaga

User guide last updated on: 19/06/2021

Relative to version 1.1.0

Introduction

Hello and thank you for purchasing *Overworld 2D Generator*.

Before starting let me briefly introduce the goal and the motivation which made me create and publish this asset.

If you instead prefer to immediately go on and understand how it works, you can jump to the "*Quick Start*" section of this user guide.

During the development of my personal project, I ran into some of the most common problems which some indie developers have to deal with, like the limited available resources and the limitations of my artistic and design skills, as opposed to my programming and organizing skills, and this while the initial design was growing in complexity and becoming more challenging. Game worlds often require a long and accurate design phase for all their levels and environments, and I witnessed the difficulty of creating one from scratch. Fortunately a different approach exists in the game industry, in particular for those games we could consider belonging to the ***sandbox*** category, like procedural worlds and levels generation, or, more generally, procedural generation.

After having considered many alternatives for my project, I decided to create myself what I needed: a complete overworld generator. And not just something which would help me generate a simple world map, but something that would integrate all the typical elements in an rpg-like game overworld. And even before starting its development, I thought it could have been a useful asset for other developers' games.

In this guide I will try, at my best, to explain what Overworld2DGenerator is, how it works and how to use it. Don't hesitate to contact me if needed.

My publisher information page at:

https://assetstore.unity.com/publishers/52808?preview=1

Overview

Overworld2DGenerator is essentially a 2D game *overworld* (or *world-map*, if you prefer the term) *procedural generator*, this is a tool which can generate, given a proper configuration, a completely random 2D world game level.

Basically it can:

- Generate the whole world's ***mainland*** and, if needed, the outer ***sea***, depending on the type of desidered overworld (for example one or more continents surrounded by the sea).
- Generate different types of ***terrains*** all over the world's land (like grasslands, forests, mountains, etc.) or even the sea, in the desidered way (more or less diffuse, more or less sparse, etc.) and with desidered conditions (some terrains at specific latitudes, like desertic or snowy ones, terrains only inside other specific terrains, etc.).
- Populate the whole world with many "***map objects***" (like cities, villages, some particular environmental elements, interactable elements like chests, etc.) in different world's positions depending on the specified conditions (some objects only over specific terrains, or only in a specific world range, etc.)

But it is not just a world-map/terrains generator, it is also a real overworld creator and manager, which helps you to create and populate an "alive" overworld, and integrate it with the final game.

Its other features include:

- Locally ***save*** a generated Overworld and ***load*** it anytime you want (for example for the same game-save)
- Dynamically and ***randomly spawn*** and manage "living" elements in the world (like npcs or enemies) or other type of objects (like collectables)
- Implementing a ***random-battle*** mechanism for when the player move in the world, based on the current terrain the player self is on
- Manage the overworld ***scene switch***, also helping you in managing the temporary or permanent game data during the changing, on events like:
    - interaction with overworld ***map places*** (cities, villages, dungeons, etc., for loading the relative area scene)
    - Interaction with ***overworld enemies*** (for loading battle scenes), or other characters
    - Random battle encounter occurrences (again for loading battle scenes)
- Define the specific sets of ***battles' enemies*** on specific overworld encounters, depending on the specific event (random battle or visible overworld enemy) and current world position
- Define a basic ***player interaction*** interface with the overworld's elements

Since Overworld Generator is a 2D world generator, it works with **2D type assets** (sprites), and can use both Tilemaps and SpriteRenderers for rendering the different parts of the world, depending on how it is configured. You can use any tilemap sprite and any 2D type asset resource you like, but with only the following limitations:

- Only **squared tiles**
- Tile names of each sprite used the given **tiles' nomenclature**

Quick Start

The required components for the overworld generation (or loading) are the following:

- **OverworldDataGenerator**: generates or load the "abstract" data of the world
- **OverworldVisualGenerator**: "physically" generate the world
- **OverworldGeneratorManager**: manages the other components

You could add all these components to the same GameObject in the scene or to different game objects. *OverworldDataGenerator* and *OverworldVisualGenerator* scene objects have to be set as fields of the *OverworldGeneratorManager* scene object though.

There are also other overworld management components you can set by other OverworldGeneratorManager fields, but they are optionals.

After that you have to configure OverworldDataGenerator and OverworldVisualGenerator *settings*, add the needed overworld ***generation assets*** to the first one, and set the ***graphical assets*** to load (sprites and tiles) in the second one.

The overworld *generation assets* are scriptable objects created through the context menu "*Create->Overworld Generator->Base Generation Data*".

For your *graphical assets* you can use your own sprites and tilesets, just ensure the ones used for world terrains rendering have the correct nomenclature (see "Assets nomenclature rules" in the following section). Alternatively you can pass your own nomenclature setting.

More of the Overworld Generator components and assets will be specified later.

A quick-prototyping template scene is already provided in the folder "*Assets/Overworld Generator/Scenes*" with the essential generator components, plus some optional components for overworld managing; *generation assets* have still to be created and passed however.

For a ready and configured scene you can instead check one of the demo scenes in "*Assets/Overworld Generator/Demo/Scenes*". As declared in *DemoNote.txt* you should add the launched scene and the scenes in "*Assets/Overworld Generator/Demo/Scenes/Loaded Scenes*" to the build settings, for testing the overworld scene switch (on places and battles) feature, but the *DemoIntro* component can do it for you on scene start.

How it works

NOTE:

I will now give a quite deep explanation about the whole overworld generation and management process: you can skip to the *Overworld 2D Generator Components* paragraph, if you just want concrete information on how to set each generation component for a quick start overview. Maybe just check the "*Assets nomenclature rules*" of this paragraph if it is not clear on how tiles assets should be named.

Overworld2DGenerator generates a whole 2D world, using a regular grid subdivision in squared units, called **tiles** (to not be confused with Unity tilemap tiles). Each tile has a prefixed size, and the world self has a prefixed tiles size, both in with and in lengths; each world tile can then be identified by an integer [x, y] coordinate, in the range of [0, world width] and [0, world length].

Each tile is then conceptually subdivided in a number of different layers, or levels: a given terrain which occupies a certain world tile, is placed on one and only one level of that tile. The number of levels is the same for all the world tiles, so it could be considered subdivided in a regular tri-dimensional grid NxMxL, and the terrain on a particular level of a tile can be accessed or set using the relative [x, y, z] coordinate.

The main *motivation* for considering multiple levels for the same tile, is the consideration of what king of graphical tileset the user can utilize for the overworld rendering: some tiles could be rendered over other ones (think for example to a forest over a grassland) and so the terrain overlapping has to be taken in account.

During the *generation process*, each terrain to be generated extends over a certain number of world tiles and for each tile always on the same level. On the same world tile, a terrain could overlap with other ones, if they are placed on different levels, but it can't extend on that tile if another terrain has already been placed on the same tile level, unless expressly specified.

The other part of the whole overworld data, are the overworld ***map objects***: these are basically those elements which are part of the world itself, like *cities*, *villages*, *isolated environmental elements* or even *treasures* visible on the world map. Like terrains these are part of the generated overworld and saved along the overworld data, but following

different generation data and rules: they are not mapped on the world tiles (*) but saved on a different collection assigned to a specific world position.

(*) NOTE:

There's actually an exception in this behaviour, since a map object can actually be saved as a terrain tile and memorized in the overworld terrain tiles' data, but, conceptually speaking, it can then be considered as an actual terrain (with no edge-transition), different from other map objects.

Overworld data generation

There are two types of generator components: the *Data Generator* and the *Visual Generator*. The latter one "physically" generates the overworld based on the overworld data already generated by the first one, basically it simply spawns and sets the graphical components needed for visualizing it, this is the tiles on the tilemaps and/or GameObjects with SpriteRenderers attached.

Before the overworld visualization anyway, the "actual" overworld generation needs to be performed; and its visualization must await the whole generation process; depending on world size settings, the number of terrains (and map objects), their extensions and other settings, it could also take a while. Obviously if the world has been already generated and saved into a file, it can be just loaded, in much shorter time.

Even before the generation anyway, the Data Generator needs to be initialized, given its settings and passed assets. The whole process of the world generation can be resumed in the following steps:

1) The overworld overall terrains' information is initialized, this also comprise the terrains' hierarchical structure information; in the same way also map objects information is initialized
2) The terrains' hierarchical structure is navigated, from parent terrains to child ones, and for each one:
    a) The terrain is expanded above its parent (or the whole mainland), eventually within a defined range and following given rules, tile-by-tile until a certain percentage is reached
    b) For each world tile position occupied (as [x, y] position), the terrain is mapped on one of the tile levels and this value is saved on the overworld data ([x, y, z] position of the grid)
    c) Eventually an optimization check is performed, for example for avoiding unwanted tiles-transitions, in one or more steps
3) After the terrains' generation, the map objects are generated, following the generation/spawning rules of each one, and saved as part of the overworld data

After this, if needed, the overworld data can be saved on a file for being loaded a second time.

In the process of loading instead, only the first one of the previous steps needs to be carried out.

Overworld Visual Generation

Once overworld data has been generated or loaded, it can be accessed and read for its effective visual generation; this task is carried out by the **Visual Generator**. Just like the Data Generator needs to be initialized, starting from its settings which define all the visualization and rendering aspects. In particular is defined the area, the "spawning" or "visualization") inside which all rendering elements (Tilemaps' tiles and SpriteRenderers) are set and outside which are unset, and how is updated; allowing the instancing of overworld elements only inside this area, avoids waste of computational resources.

Each time the visualization area needs to be updated (for example because the camera moved), find the tiles which need to be updated and for each of these, for each tile level, set or unset the relative graphical asset to be used. Since this is a 2D world generator, these assets are Unity Tilemaps' tiles (only for terrains) and SpriteRenderer objects (for terrains, even if these tilemaps are preferables, and map objects). The name of the asset to be set depends on the binding between the terrain on a particular tile level and the associated data found in Terrain Tiles Data assets, and by its content. On which tilemap sets each tile or what *sorting layer* and *sorting order* set for each SpriteRenderer, depends instead on Visual Generator settings and default Terrain Tiles Data assets, used for the generation, default parameters.

What this generator does, regarding the generated terrains spawning, is also detect the correct graphical asset (sprite or tile) to use, each time it has to set one for a particular world tile, on a particular level, depending on the (eight, if not on the world borders) nearby tiles. Basically it detects if the tile is a terrain *border* tile or not, checking if at least for a nearby tile there are different values on the same level; a tile with a border on a level it's said to contain an *edge-transition* on that level. When an edge-transition is detected a suffix is appended to the base asset name (both for tiles and sprites) and the correct asset is searched.

Assets nomenclature rules

Terrain tile assets (tilemaps' tiles and sprites) have to follow the correct nomenclature, at least for terrains with borders/edge-transitions: this generally consists in appending a suffix to the name of the base tile asset (the one without edge-transitions), depending on the needed transitions/borders. You can follow the default nomenclature or define a custom one.

Default nomenclature

The default nomenclature is the following:

- If there're not edge-transitions/borders the base tile name is used
- If there are edge-transitions/borders a suffix is appended for each of the nearby tile (in total each tile has 8 nearby tiles) with a different type of terrain, in the following order:
    - For the bottom-left nearby tile '***_SW***' is appended
    - For the bottom nearby tile '***_S***' is appended
    - For the bottom-right nearby tile '***_SE***' is appended
    - For the left nearby tile '***_W***' is appended
    - For the right nearby tile '***_E***' is appended
    - For the top-left nearby tile '***_NW***' is appended
    - For the top nearby tile '***_N***' is appended
    - For the top-right nearby tile '***_NE***' is appended

The order in which suffixes are appended is important, so for example you could have a *'SW_S_W'* suffix but not *'SW_W_S'*.

For example:

- for tiles with western edge border ***'_SW_W_NW'*** has to be appended
- for tiles with western edge and northern edge border and north-west outer corner ***'_SW_W_NW_N_NE'*** has to be appended
- for tiles with north-west inner corner border ***'_NW'*** has to be appended

Custom nomenclature

If you want to define a custom nomenclature, for example if you already have a set of tiles with their own edge-transition nomenclature which you don't want to change, you can create an "*TileSuffixConfiguration" asset* and set it for the Visual Generator. In this way you define a custom global rule, but you can also redefine it for each particular terrain, creating and setting for them other *TileSuffixConfiguration* assets.

Tip:

If you don't really use a custom one, you will probably just use the default nomenclature rule and rename the assets as needed.

Tip2:

You can use the "Tiles Sprites Renamer Window", in windows menù, to easily rename a set of sprites, see the relative section.

You can check for an example the used resource tilemaps in the Demo folder.

NOTE:

Most tileset assets don't support all kinds of edge-transition and generally only a subset of possible combinations are supported. North, south, west and east edges, and north-west, north-east, south-west and south-east corners are generally considered: these are the *conventional/standard edge-transition*s. A '_N_S' transition can instead be considered unconventional.

To solve this, you can specify which terrains support a non-standard set of edge-transitions, and which ones don't. The generator will consequently adjust the generated terrains to avoid unsupported transitions, clearing terrain tiles when needed.

A good example of a tileset supporting all usable edge-transition combinations can be, for example, any *RPGMaker-like* tilesets.

Overworld overall management

Even if the *Data Generator* and the *Visual Generator* could be used separately, their function is actually controlled by a supervisor component which also coordinates all other overworld components. The goal of the Overworld Generator in fact, is also to provide a full management of an entire overworld, not just its generation/initialization.

This **Overworld Manager** manages the overall generation, loading and unloading of the overworld: it initializes both the generators, in the correct order, proceeds with the overworld generation or loading and enables the visualization, waiting if needed for the completion of each step.

More on the **OverworldGeneratorManager** component later.

Overworld Generator Components

Following the list of all the components used, order by relevance.

NOTE: most of these components use a **custom editor**, so if you want, for any purpose, to edit any of them you should probably edit the relative editor too.

OverworldGeneratorManager

It manages the interaction with other generation components, among which Data and Visual Generators, the loaded overworld scene data, provide support with components initialization and the initialization of the whole overworld itself; it also manages other utility stuff like player game object instancing.

This component is basically the entry point for the overworld generation and management. Ideally it should be the only one to interact with, when this asset is integrated with the actual final game.

Component fields

Overworld generators

Data Generator: the scene object with the *OverworldDataGenerator* component, this is needed at initialization.

Visual Generator: the scene object with the *OverworldVisualGenerator* component, this is needed at initialization.

Find Generators In Scene: if not already set both the generators can be automatically found in the scene at start.

Overworld scene management

Scene Loader Game Object: a scene object with a component which implements the *ISceneLoader* interface, needed for overworld scene-switching. (*)

Scene Data Holder Game Object: a scene object with a component which implements the *ISceneDataHolder* interface, not needed to be set but in case you have to specify at least the SceneDataHolderName field and manager will spawn a GameObject with a base implementation at runtime.

Scene Data Holder Name: name of the SceneDataHolder GameObject to be spawn

Scene Data Holder Template: if SceneDataHolder has to be spawn at runtime, an instance of prefab passed in this field can be used, instead than an empty GameObject; if it hasn't a component implementing the ISceneDataHolder interface, a default implementation will be attached at instantiation time.

Overworld asset/resources management

Asset Bundle Manager Game Object: a scene object with a component which implements the *IAssetBundlesManager* interface, not needed to be set but the manager will set a runtime implementation if needed.

Create Asset Bundle Manager If Needed: asset bundle manager will be automatically created on need if not set (on Asset Bundle assets loading).

Save data management

Scene Data Manager Game Object: a scene object with a component which implements the *ISceneDataManager* interface.

Save Data Manager Game Object: a scene object with a component which implements the *ISaveDataManager* interface.

Save Data Manager Object Name / TAG: name of the scene object with the component: if the reference to the object is not directly set, this can be used to find it in the scene. Object's name is used first, if set, then the object's TAG.

Save Data Manager Instantiate Template: Prefab with a component which implements the *ISaveDataManager* interface: if no scene object with the needed component is passed or found, if this is set it will be used to instantiate one at runtime.

Optional components fields

Inventory Manager Game Object: a scene object with a component which implements the *InventoryManager* interface.

Inventory Manager Object Name / TAG: name of the scene object with the component: if the reference to the object is not directly set, this can be used to find it in the scene. Object's name is used first, if set, then the object's TAG.

Inventory Manager Instantiate Template: Prefab with a component which implements the *InventoryManager* interface: if no scene object with the needed component is passed or found, if this is set it will be used to instantiate one at runtime.

Interface management

Overworld GUI Game Object: a scene object with a component which implements the IOverworldGUI interface.

Input Source Game Object: a scene object with a component which implements the *IInputSource* interface.

Overworld Player Character

Player Template: a prefab of the GameObject representing the player to be controlled in the overworld, set it if you want the manager to spawn it for you

Instantiate Player: check this if you want the player object to be automatically spawn just after the overworld generation or loading, in this case *PlayerTemplate* field has to be set

Player Map Object Starting Position: name of the map object (the type of the map object, it is the '*map object name'* field of map objects data asset's entries) at which position the player will spaw the first time (on generation, not loading): if set, the component will search for map objects of this type in the world and use the position of the first one found (if any).

Player World Starting Position: position at which position the player will spawn the first time (on generation, not loading); if '*Player Map Object Starting Position*' is set this position will be an offset of the found map object's position.

Overworld Active Objects

Remove Overworld Objects Outside Generation Area: overworld objects outside the *visual* generation area (specified by the VisualGenerator component) are removed on each check (which also depends on the VisualGenerator component). The objects removed are the ones registered as active overworld objects on the Manager components itself (see the relative method below).

Saving and startup settings fields

Default Save File Folder: folder path of the overworld save file location, for when the generated overworld is saved, this is relative to the *persistent data path*.

Default Save File File: name of the overworld save file, for when the generated overworld is saved, this is the base filename and a timestamp will be appended to it that is saved by calling the opposite method.

Default Save File Extension: extension of the overworld save file, for when the generated overworld is saved.

Startup Action: enum field indicating what the manager will do at scene startup:

- *NOTHING*: the manager will not generating nor loading any overworld, the overworld generation or loading has to be managed through custom code
- *CREATE*: the manager will generate a new overworld, without saving it; bear in mind, in this case, the manager will not keep overworld data on non-additive loading of another scene, and it will need to be re-generated when reloading the overworld scene.
- *CREATE_AND_SAVE*: the manager will generate a new overworld and save it in the specified default file path (DefaultSaveFileFolder + DefaultSaveFileFile + DefaultSaveFileExtension).
- *LOADING*: the manager will load an already generated overworld, saved in the specified default file path (DefaultSaveFileFolder + DefaultSaveFileFile + DefaultSaveFileExtension).
- *USE_SAVE_DATA_MANAGER_SETTINGS*: the SaveDataManagerComponent will be used for detecting if generating or loading the overworld, as for reading the name of the file to use.

Once Overworld Generator is fully integrated with the final game, this field should be probably set to *USE_SAVE_DATA_MANAGER_SETTINGS*, with a component able to managing the save data and detecting the player's choice of a new or loaded game, or to *NOTHING* and manage everything through custom code.

Debug settings fields

Debug Log Game Object: a scene object with a component which implements the IDebugLog interface.

Note

Components objects fields use this *wrapping pattern*: you just pass a generic scene object and the manager will search for a component implementing the needed interface; if the component is found it will be used, otherwise the passed object will be discarded and if needed a default component will be used.

These components can also be set with the following public properties:

IAssetBundlesManager -> **OverworldGeneratorManager.AssetBundleManager**

ISaveDataManager -> **OverworldGeneratorManager.SaveDataManager**

ISceneLoader -> **OverworldGeneratorManager.SceneLoader**

IInventoryManager -> **OverworldGeneratorManager.InventoryManager**

ISceneDataHolder -> **OverworldGeneratorManager.SceneDataHolder**

IOverworldGUI -> **OverworldGeneratorManager.GUI**

IInputSource -> **OverworldGeneratorManager.InputSource**

IDebugLog -> **OverworldGeneratorManager.DebugLog**

For most of them a base implementation is provided and ready to use but you can create and use whatever interfaces' implementation you prefer, for a better integration with the final game. Since these components are optionals you are neither forced to set and use any of them.

For more info on each one see "Secondary management components" section below for a quick overview, or check the demo scenes.

NOTE

Since all the overworld initialization and generation/loading is done on *Start()* method (if not managed by custom code), if needed it's suggested to set them in the *Awake()* method.

Component public methods

You can interact directly with the OverworldGeneratorManager component (and so, with the overworld itself) by using the following public methods, for example if you want to define a custom overworld starter, for any purpose.

CreateNewOverworld*(bool* ***useDefaultSave****, bool* ***appendDatetimeToFilename****)*: generate a new overworld, if first parameter is set to true the save file path will be obtained by default save settings fields, otherwise it will not be saved, if second parameter is set the current timestamp will be appended to save file name; the folder path is considered relative to the persistent data path.

CreateNewOverworld*(string* ***fileFolder****, string* ***fileName****, bool* ***appendDatetimeToFilename****, string* ***fileExtension****)*: generate a new overworld, passing save file folder, file name, file extension and if timestamp has to be appended to the name; the folder path is considered relative to the persistent data path.

CreateNewOverworld*(string* ***fileFolder****, string* ***fileName****)*: generate a new overworld, passing save file folder and file name (comprehensive of extension if any), if one of them if null overworld will not be saved; the folder path is considered relative to the persistent data path.

CreateNewOverworld*(string* ***filePath****)*: generate a new overworld, passing the whole file path, if null overworld will not be saved; the folder path is considered relative to the persistent data path.

LoadOverworld*()*: load the overworld using the save file path will be obtained by default save settings fields.

LoadOverworld*(string* ***saveFilePath****)*: load the overworld saved at the specified save file path.

IstantiateOverworldPlayer*()*: instantiate the player template object in the world, if not automatically instantiated you can use this

SetOverworldPlayer*()*: directly set the in game player in the world, instead of instantiating through a player template

SetOverworldEnabled*(bool* ***enabled****, bool* ***setOverwordCamera****)*: set the whole overworld enabled or disabled (basically shown or hidden and with all its element active or inactive), if the second parameter is set also the game Camera which is rendering the overworld scene will be enabled or disabled; it is used, for example, for hiding the whole overworld when loading another scene in additive mode.

AddOverworldActiveObject/RemoveOverworldActiveObject*(GameObject* ***overworldObject****)*: add/remove an object to the *active overworld objects* of the Manager: the objects in this list are enabled and disabled (so shown or hidden) with the overworld itself. If you only used GameObjects with a component extending ***OverworldObject*** class (see relative section), they will automatically call this method on awake.

These are basically the methods which establish the overworld manager interface, the other public methods are meant to be used by the other overworld's components for the whole functioning, but you can use them anyway if you understand how they work.

Component events

OnOverworldEnable(bool ***overworldEnabled***): triggered on each call of ***SetOverworldEnabled*** method, so each time the overworld is enabled or disabled, and useful for managing default or custom logic which needs to be disabled as well.

OverworldDataGenerator

This component generates, loads and more generally manages the overworld generated data, hiding the complexity through an interface for accessing the data itself. It's settings and data fields need to be set both in the case of a world first generation and world data loading.

Component fields

World general settings

Basic settings of the world generation, in particular you set the size of the generated overworld, the generation seed and the data format in which you want to save it. Remember to give a proper data format, if you define a lot of terrain types, even if you would just probably need to leave the default value (BYTE) which can handle up to 254 terrains.

World Width: width of the generated overworld (in terrain units/tiles).

World Length: length of the generated overworld (in terrain units/tiles).

UseRandomSeed: if the overworld generation seed used should have a random value; if not using a random value consecutive generations generate the same results.

GenerationSeedValue: value of the generation seed, if not let to have a random value.

World Tile Data Unit: allow to select the unit data type for each single terrain unit (this is for each single terrain tile); since the overworld terrain data is represented by a tri-dimensional grid of tiles NxMxL (tile's X-position, tile's Y-position and tile's level) the number of possible values each tile can have, and so the maximum number of terrain tile types you can have, depends on the max value of that type (precisely it will be *Type.MaxValue* - 1, since one value is for meant for the Undefined/Unset value). You should obviously select the smallest possible value you can have or you will have a waste of memory. It can have the following values:

- *BYTE*: the overworld terrain data is a tri-dimensional grid of **byte** and the number of usable terrain tile types are 254
- *UNSIGNED_SHORT*: the overworld terrain data is a tri-dimensional grid of **ushort** and the number of usable terrain tile types are *UShort.MaxValue* - 1
- *INTEGER*: the overworld terrain data is a tri-dimensional grid of **int** and the number of usable terrain tile types are *Int.MaxValue* - 1

NOTE: In reality, the overworld terrain data is internally a one-dimensional array suitably sized and not a tri-dimensional matrix, but the OverworldData class masks this and it's interface allows its access as if it were a real tri-dimensional grid.

Terrain Tile Data Assets: list of the used terrain tiles data assets (see the relative section).

Mainland generation settings

Settings on how to generate the whole world mainland. The Generated Overworld Type general setting allows you to select for different types of overworlds; other settings depend on it and define a more fine-tuned generation.

Generated Overworld Type: base type of the overworld generated, many following parameters depend on this one:

- *ONLY_LAND*: generate only land, without the sea, all sea-related parameter are hidden if this is selected
- *LAND_WITH_SEA_BORDER*: generate land with sea on the specified borders
- *ISLAND*: generate a single continent-like landscape at the centre of the world and surrounded by the sea
- *MULTIPLE_ISLANDS*: generate multiple continent-like landscapes, at the specified position and surrounded by the sea
- *CUSTOM*: use a custom generator for a custom generation

South/North/West/East Sea Border: if Generated Overworld Type set to *LAND_WITH_SEA_BORDER*, the flags set on which border is the sea

Mainland World Borders Distance: minimum distance of any land from the world borders, defines the external part of the world occupied only by sea where the mainland cannot extend.

Use Different Border Distance For Each Edge: the mainland world border distance is defined with different values for each edge.

Mainland South/North/West/East World Borders Distance: the mainland world border distance defined for each world edge.

Land Generation Algorithm: type of generation algorithm used:

- *FILL_PERCENTAGE*: a simple fill algorithm, fill the available space until a given percentage value is reached, using a cohesiveness factor for the filling placement
- *PERLIN_NOISE_WITH_FALLOFF*: use the Perlin Noise algorithm for the land generation, applying a threshold value and a falloff on the world borders, for allowing an isolated landmass.

Land Percentage: if *FILL_PERCENTAGE* algorithm selected, percentage of the world area (in tiles), inside world external sea borders, occupied by land.

Land Cohesiveness: if the FILL_PERCENTAGE algorithm is selected, the cohesiveness factor of the whole land, the higher it is the lower the probability it will be fragmented in many island-like parts.

Noise Scale Value: if *PERLIN_NOISE_WITH_FALLOFF* algorithm is selected, the scale applied to the perlin noise map, the bigger it is the less the generated surface will be jitted.

Noise Threshold Value: if *PERLIN_NOISE_WITH_FALLOFF* algorithm selected, threshold value for applying the sampling on the Perlin-Noise generation map.

Falloff Function Type: if *PERLIN_NOISE_WITH_FALLOFF* algorithm is selected, the falloff function type for applying the falloff on the Perlin Noise generation map.

- *LOGISTIC_SIMPLE*
- *LOGISTIC_ADVANCED*

Falloff Factor: if the PERLIN_NOISE_WITH_FALLOFF algorithm is selected, the falloff factor is applied to the falloff function.

Falloff Factor 2: if *PERLIN_NOISE_WITH_FALLOFF* algorithm and *LOGISTIC_ADVANCED* falloff function selected, the second falloff factor applied.

Remove Inner Sea Regions: remove the regions of the sea generated inside land regions, this is the regions not linked to the external parts of the sea.

Inner Sea Regions Custom Remover: define a custom inner sea region remover object, useful, for example, for custom land generation

Base Sea Tile: name of the tile representing the default sea terrain; at generation start all sea portions of the world will be set to this.

Base Land Tile: name of the tile representing the default mainland terrain; at generation start all mainland portions of the world will be set to this.

Base Land Terrain Name: the name that will be associated to the base land terrain, this doesn't need to be defined in an asset

Base Sea Terrain Name: the name that will be associated to the base sea, this doesn't need to be defined in an asset

Generate World Border Terrain: if you want to define a terrain extended on the external border of the world; this could serve as a barrier preventing anything from going out of the boundaries.

World Border Terrain Tile: terrain tile name used for the world external border

World Border Terrain Size: size (width) of the world external borders in tiles number

Generated Terrains settings

Settings for all the terrains generated as children of mainland terrain (or the base sea "terrain"). The terrains which need to be generated are to be specified, so as general rules for terrains' generation.

Terrain Generation Data Assets: list of the used terrain generation data assets (see the relative section).

Use Climate Zone: if you want to use climate zones.

Base Land Terrain Name: if climate zones are NOT used, the name of the terrain (not of the terrain tile) used as base for the mainland generation.

Climate Zones: if climate zones are used, list the used climate zones ordered from southmost to the northmost. For each climate zone entry you specify the following:

- *Climate Zone Name*: name assigned to the climate zone, just for representative purpose.
- *Climate Zone Relative Range*: the climate zone latitude extension size, relative to the overworld's mainland extension length and the other climate zones' range; for example if a climate zone has double extension than any other you can set this to 2 and all the other ones to 1.
- *Climate Zone Terrain*: name of the main terrain (at the top of hierarchy) for the climate zone.
- *Use Associated Terrain Tile*: if the set will be used the first found Terrain Tile associated with the set Climate Zone's terrain, between all the defined ones, as base for the climate zone, otherwise it needs to be directly specified.
- *Climate Zone Tile*: name of the terrain tile used as base for the climate zone.
- *Define Climate zone sea*: define a climate zone specific "terrain" also for the sea inside the climate zone
- *Climate Zone Sea Terrain*: if Define Climate zone sea selected, the same as for climate zone land terrain
- *Use Associated Sea Terrain Tile*:if Define Climate zone sea selected, the same as for climate zone land terrain tile
- *Climate Zone Sea Terrain Tile*:if Define Climate zone sea selected, the same as for climate zone land terrain tile

Terrains: the list of all terrain to generate, it is the terrains child of climate zone terrain or the main terrain, as specified in terrains generation assets; if GenerateAllDefinedChildrenTerrain is set, all these terrains are generated.

Generate All Defined Terrains: all the terrains in the Terrains Generation assets are generated, in this case the terrains' to generate list is not used.

Max Tile Transition Correction Passes: Max number of passes for applying tile transition fix, for tiles specified to not use non-standard tile transition, in *TerrainTilesData* asset. For each pass the fix stops if no more tiles are corrected. More passes tend to grant better results at the cost of worse performances.

Generated Map Objects settings

Settings for all the map objects which need to be placed in the world (on the land or even in the sea if specified). The map objects which need to be generated are to be specified, so as general rules for map objects' placement.

Map Objects Data Assets: list of the used map objects' generation data assets (see the relative section).

Map Objects: the list of the map objects to generate; if GenerateAllDefinedMapObject is set, all the map objects defined in map object generation assets are generated.

Generated All Defined Map Objects: all map objects defined in the Map Objects Generation assets are generated, in this case the map objects' to generate list is not used.

Map Object Default Avoid Terrains: the defined terrain TILEs will be avoided by default for the generation and placement of any map object; single map objects can define additional terrains to avoid (see Map Objects Data asset section).

Map Objects Default Clear Terrains: the defined terrain TILEs will be cleared by default of any map object's generation and placement, if at its position; single map objects can define additional terrains to avoid (see Map Objects Data asset section).

Map Objects Max Placement Tries: Maximum number of tries for the placement of any map object, if 'Force Placement' has NOT been set (see Map Objects Data asset section), following all the placement rules.

Map Objects MaxForce Placement Tries: Maximum number of tries for the placement of any map object, if 'Force Placement' has been set (see Map Objects Data asset section), following all the placement rules.

Find Feasible Map Object Tiles For Default: for default, before trying to place any map object type in the world, the generator will find all the possible tiles over which it can be placed; in this way placement tries will be avoided.

NOTE

The *Find Feasible Map Object Tiles* setting can be specified as a general default setting for all map objects' generation or for each single map object, in the generation asset, but probably you should only use it in the latter way: while in fact it allows to avoid useless

placement tries, it require a pre-calculation time for each type of object to be place, for finding the feasible placement positions.

Tip:

Use the *Find Feasible Map Object Tiles* setting maybe only for map objects which have strict placement rules and/or are spawned in great quantity (Spawn Count), otherwise going with placement tries could be a better option.

OverworldVisualGenerator

This component is responsible for visualizing the generated world itself, which comprises overworld terrains and map objects, using Tilemaps and SpriteRenderers, depending on its configuration. All the generated elements are always located inside the generation/"spawning" area, for an efficiency purpose, so only a part of the world is visualized.

This area can be automatically updated on the need, every time the overworld rendering camera changes position or its field of view, also if it can be entirely managed by code; It's suggested to set at least the first option though. It should also be set a bit bigger than the camera render area size, for avoiding pop-up effects when the camera is moving.

Component fields

Generation AreaSettings

General settings about the *visualization area*, which is the part of the overworld currently visualized by the player and inside which the visualization elements are spawned (and outside are destroyed)

Overworld Render Camera: camera used to render the overworld scene. The visualization area (both its position and size and size if specified) will depend on it (position and view frustum).

NOTE: for a 2D rendered world you should use an orthographic camera!

Use Main Camera If Not Set: if the camera field is not set, the main game Camera will be used.

Default Generation Area Size: starting size of the Visual Generator area size, if the component is not set to update with camera render area it will remain of this value; in this case it is suggested to be a bit bigger than camera render area.

Update Area With Camera Position: the visualized will be updated on camera position change (this is the suggested configuration).

Update Area With Camera Size*:* the visualized will be updated on camera view size change.

Camera View Area Size Ratio: if the area is updated with the camera view, how much bigger it will be (default 1.5 times bigger, in width and length).

Out Of Area Objects Check Distance: a check on the overworld objects which are outside the visualization area will be performed each time the area position changes to the

specified distance; this will determine what objects are removed because outside such area (depending on *OverworldGeneratorManager* component settings).

Terrain Tiles Layout Settings

General settings about the world's tiles' layout: all world's tiles should have the same layout settings (so the same tile size and offset).

Tiles Grid Layout: grid layout of the used tilemaps, if not set it will be automatically searched in scene; only one grid layout can be used by all tilemaps

Tiles Position Offset: position offset, in world-units, for each terrain tile; this should be set based on all tileset sprites' pivot point and all tilemaps' anchor point. If all sprites have a CENTER pivot point and all tilemaps have the default (0, 0) anchor point, you should leave it to (0.5, 0.5) value. It's also used for world-to-tile position conversion.

Tiles PPU Size: size in point-per-unit of each world tile unit; all used render tiles are supposed to be of this size. It's also used for world-to-tile position conversion.

Terrain Tiles Naming Settings

These settings define the *tiles' nomenclature*, this is the name each tile asset (Sprite on Tilemap's Tile) should have, depending if and which edge-transitions it represents. See the relative section for more information.

Use Custom Transition Suffix Configuration: set if you want to use a custom nomenclature, otherwise the default nomenclature rules are used

Suffix Configuration Assets: if you set for a custom nomenclature you pass here an *TilesSuffixConfig* asset (see the relative section).

Default Tile Transition Prefix: suffix-prefix to be appended to any edge-transition suffix, if tiles assets have one; for example if the base tile name id *'Tile'* and transition ones are *'Tile_borders_SW'*, *'Tile_border_S'*, instead of just *'Tile_SW'*, *'Tile_S'*, etc.

Tiles Dictionary Assets: list of TilesDictionary assets used (optional, see relative asset section).

Visualization through Tilemaps settings

These are the settings relative to all world elements meant to be rendered through a *Tilemap* component.

Used Tilemaps: list of all in-scene tilemaps used by the Visual Generator, for each entry the tilemap entry a name needs to be specified, which will be used for the association with each generated element using tilemaps. Alternatively the component can automatically find and use all the tilemaps in the scene, using as association the GameObject's name.

Find And Use All Tilemaps: the components automatically find all tilemaps.

Tiles Resources: tiles assets to be loaded for the component to use them at runtime. (*)

Terrain Tiles-Tilemaps Bindings List: the list of the binding between the terrain tiles (defined in terrain tiles assets) and the used tilemaps.

Use Default Terrain Tiles - Tilemaps Bindings: the default terrains - terrain tiles associations, considering the default associated terrains, and the default Tilemap's name will be used for each defined terrain tile (see Terrain Tile Data asset section); in this case, if *Override Default Bindings* is not set, tilemaps' binding list is not used

Override Default Terrain Tiles - Tilemaps Bindings: in addition to using the default Tilemaps bindings you can also define additional bindings or override existing ones.

Visualization through Sprite Renderers settings

These are the settings relative to all world elements meant to be rendered through a *SpriteRenderer* component.

Sprites Resources: sprites assets to be loaded for the component to use them at runtime. (*)

Sprite Render Template: prefab with a SpriteRenderer component to be istantiated for each terrain tile rendered through a SpriteRenderer.

Terrain Tiles Sprite Root: all instantiated terrain tiles sprite renderers will be instantiated as child of this object.

Terrain Sprites Root Name: if not set, TerrainTilesSpritesRoot can be found by its name.

Terrain Tiles-Sprite Renderers Bindings List: the list of the binding between the terrain tiles (defined in terrain tiles assets) and the used sprite renderer' sorting layer.

Use Default Terrain Tiles - Sprite Renderers Bindings: the default terrains - terrain tiles associations, considering the default associated terrains, and the default Sorting Layer will be used for each defined terrain tile (see Terrain Tile Data asset section); in this case, if *Override Default Bindings* is not set, tilemaps' binding list is not used

Override Default Terrain Tiles - Sprite Renderers Bindings: in addition to using the default Sprite Renderers bindings you can also define additional bindings or override existing ones.

Update Map Terrain Tiles Sprite Sorting Sorting Order With Position: terrain tiles' SpriteRenderer's layer sorting order will be automatically updated on generation, based on the map object y - position

Terrain Tiles Sprite Sorting Sorting Order Base Offset: offset to add to the generated Terrain Tiles' SpriteRenderer's layer sorting order

Map Objects Sprite Root: all instantiated map objects sprite renderers will be instantiated as child of this object.

Map Objects Root Name: if not set, MapObjectsSpritesRoot can be found by its name.

Map Objects - Sprite Renderers Bindings List: the list of the binding between the terrain tiles (defined in terrain tiles assets) and the used sprite renderer' sorting layer. If "*Use Default Map Objects Sprites Bindings*" is set, for each entry in the terrain tiles assets, the default tilemap name will be used and the relative tilemap will be searched.

Use Default Map Objects - Sprite Renderers Bindings: the default terrains - terrain tiles associations, considering the default associated terrains, and the default Sorting Layer will be used for each defined terrain tile (see Terrain Tile Data asset section); in this case, if *Override Default Bindings* is not set, tilemaps' binding list is not used

Override Default Map Objects - Sprite Renderers Bindings: in addition to using the default Sprite Renderers bindings you can also define additional bindings or override existing ones.

Update Map Objects Sprite Sorting Sorting Order With Position: map objects' SpriteRenderer's layer sorting order will be automatically updated on generation, based on the map object y - position

Map Objects Sprite Sorting Sorting Order Base Offset: offset to add to the generated Map Objects' SpriteRenderer's layer sorting order

Assets Resources fields

Both for Tilemap and Sprite Renderer components settings, you have to specify Assets Resources: these fields allow you to specify where and how to load the relative assets at runtime. These assets can be loaded both as Unity Resources (contained in a "Resources" folder) or as a part of an Asset Bundle.

Source Type: the source type of the assets and how to load them:

- RESOURCES: the assets are loaded from a Resources folder
- ASSET_BUNDLE: the assets are loaded from an Asset Bundle

Resources Folder: for Resources source type, the folder inside a Resource folder containing the assets to load; it's not needed to specify the full relative path, it's just needs to be an unique name inside any Resource folder

Asset Bundle Name: for Asset Bundle Source type, an unique name assigned to the Asset Bundle self

Asset Bundle File: for Asset Bundle Source type, the path to the Asset Bundle file from which the assets are loaded, the folder it is relative to depends on the following field

Asset Bundle Path Folder: for Asset Bundle Source type, the folder in which the Asset Bundle is contained, you can select:

- *DATA_PATH_FOLDER*: the asset bundle file path relative to the game data path folder, which depend on the current platform (see Application.dataPath of Unity manual)
- *STREAMING_ASSET_FOLDER*: the asset bundle file path is relative to the streaming asset folder (see Application.streamingAssetFolder of Unity manual)
- *NONE*: the asset bundle file path is not relative to any particular folder and can be anywhere in the current system, in this case the absolute path need to be specified for Asset Bundle file

Load All Asset: all assets in the specified folder or asset bundle are loaded, otherwise single assets need to be specified

Assets Names: list of the names of the assets to load

Include Name Suffixes: for each name all the combinations with transition suffixes will be searched too, so just the base tile (with no edge-transitions) asset name need to be specified for each tile type

Only Standard Suffixes: only the suffixes for standard edge-transition combinations will be considered, see Asset Nomenclature Rules section for clarifications

NOTE:

for assets loaded from an Asset Bundle, the *AssetBundleManager* component is use, which should be added in scene and set to the OverworldGeneratorManager component, but it will be automatically created and set if needed otherwise; the Asset Bundle Name field is needed for the component to keep track of all loaded Asset Bundle (and loading or unloading them on need).

Secondary overworld management components

These are the optional components which could be present in the overworld scene, they have to be set in the OverworldGeneratorManager fields but not necessarily; if needed a default implementation distance is always added.

These fields follow the pattern of generic objects with a generic component: the component needs only to implement an interface, not to extend a particular MonoBehaviour, so any custom one can be used and added to a generic Game Object.

For quite all of these a generic "base" implementation has already been created.

SceneDataHolder

A component implementing the ***ISceneDataHolder*** interface. It holds the overworld scene data and keeps it during scene changing (so *DontDestroyOnLoad* should be called on). It's automatically created if not set.

The data informations it keeps are the following:

- The current overworld filename, for reloading the overworld in overworld scene
- The current player position in world
- The current place information, if a particular place/location scene has been loaded
- The current overworld battle (if any happened) informations

SceneLoader

A component implementing the ***ISceneLoader*** interface. It is responsible for other scenes loading from the current overworld scene. Generally, during the game, a new scene is loaded from the overworld when:

1) The player enters into a particular place/location, in this case the relative area scene should be loaded, based on the place type and the specific place.
2) An overworld battle happens (very common case in rpg games), in this case the scene to load is based on the specific terrain(s) the player is on and/or the enemy who "touched" the player, if not a random battle encounter of course.

Since Unity scene can be loaded in two possible ways, this is in an additive and non-additive way, the component can be configured to load in each of the two previous cases. The base implementation component is configured for loading "place" scenes in a normal way and "battle" scenes in an additive way; in this way temporary map elements (like visible enemies) are kept after battles.

SaveDataManager

A component implementing the ***ISaveDataManager*** interface. It manages the “overworld save data”, this is the data of the world which varies during the game, like opened chests, collected unique items, and whichever is needed for your game, and which is different from the *generated overworld data*.

Basically it keeps track of all map objects’ status, and eventually of the collectable content for “containers” type map objects. The idea is to get and set the needed information by: map object’s type ID, map object’s specific ID and status ID / item list ID (by default a container contains a single list of items, by default ID ‘0’, but for eventual class extensions the opportunity to have multiple item lists is provided).

It should also keep the information between scenes' switches (in a non-additive way) and, eventually, provide for its persistent saving, between different game sessions.

SceneDataManager

A component implementing the ***ISceneDataManager*** interface. It manages the “overworld scene data”, which unlike the saved data is not kept between scenes. It can be used for data which can be wiped on overworld scene unloading, like a particular collectable or enemy spawn.

InventoryManager

A component implementing the ***IInventoryManager*** interface. Basically the management of the player’s inventory, for managing the picking up of every collectable in the world. It’s obviously useful only if the eventuality, in the game, of picking up items in the overworld exists. It should as well keep the information between scenes' switches.

AssetBundleManager

A component implementing the ***IAssetBundleManager*** interface. It manages the loading, unloading and retrieving of already loaded asset bundles. If asset bundles are used for, instead of Resources folders, for loading assets, it needs to be set; if not a default one will be instantiated and set at runtime anyway.

OverworldGUI

Any component implementing the ***IOverworldGUI*** interface. The base overworld GUI, used in different cases, for example for indicating the current location, a dialog message or a confirmation.

InputSource

Any component implementing the ***IInputSource*** interface. Used, eventually, by the OverworldGUI or other controls, like the Player and Camera controllers.

DebugLog

Any component implementing the ***IDebugLog*** interface. A custom log, if not set, the OverworldGeneratorManager will use the default Log.

Overworld spawns management components

These are util components, added to the overworld scene, for random spawns in the overworld.

RandomPositionObjectSpawner

This component is used for randomly spawning objects at random positions inside the VisualGenerator area. It defines a list of objects to be spawned, and for each one the list of terrain when they can spawn.

Spawned object should be an extension of the OverworldObject, like OverworldCharacter (for example enemies or npcs) or OverworldCollectDrop, which implement IOverworldSpawned interface, but they can also be used generic objects with components implementing the IOverworldSpawned interface.

RandomBattleSpawner

This component is for creating random battle encounters, based on the player's current position and current terrain it is on. Obviously, for loading the battle scenes, it needs the SceneLoader component to be set. It defines a list of terrains' names and for each one the battle encounter chance, this is the chance for a battle encounter to happen when the player is over it.

Battle scene utility components

These components are meant to be used in a battle scene, for determining a possible group of enemies which should spawn during a battle encounter, caused either by an overworld enemy or by a random encounter

BattleEnemyGroupEnemyBinder

Used for battle scenes triggered by *overworld enemies' encounters*. It defines a series of bindings between overworld enemies' names and the possible battle groups.

It defines two lists for bindings: one list of *OverworldBattleEnemyGroupEnemyBinding* assets, and one of direct binding definitions; you can use one of the two or just both.

BattleEnemyGroupTerrainBinder

Used for battle scenes triggered by *random battle encounters*. It defines a series of bindings between terrains (terrain tiles' names), on which random encounters can happen, and the possible battle groups.

It defines two lists for bindings: one list of *OverworldBattleEnemyGroupTerrainBinding* assets, and one of direct binding definitions; you can use one of the two or just both.

Other utility components

CameraController

A basic camera controller for controlling the camera rendering the overworld. It allows for free camera movement or to automatically follow the player character instantiated by OverworldController.

PlayerController

A basic player controller for controlling the player character objects. It supposes an OverworldPlayerCharacterComponent is attached to the same GameObject, since it uses its methods for moving it.

DebugTerrainTileInspector

With this component in your scene, you can inspect the terrain values, for all levels, of any world tile in a given position: simply set the inspect camera (probably the overworld render camera), the input key and where to print the log information. On the given input the world tile over which the mouse cursor currently is (in inside the world bounds) will be read and terrains' information will be printed.

This is basically a debug-only component.

Overworld Generator Data Assets

Following the list of all the components used, order by relevance.

Data assets are scriptable objects passed to different components and used for the overworld generation and management. They are created using the relative context menù.

NOTE: most of these scriptable objects use a **custom editor**, so if you want, for any purpose, to edit any of them you should probably edit the relative editor too.

Base generation data assets

TerrainsData

Context menù: Create -> Overworld Generator -> Base Generation Data -> Terrains Data

This asset contains information about the terrain's base generation. Using this data the Data Generator components can create the information on how terrains are hierarchically organized and proceed with the first step of the generation. Ideally the output of this data is the information, for each generated terrain, how it extends over the world.

For each element the following fields are defined:

Terrain Name: the name of the terrain to generate

Has Parent Terrain /Parent Terrain Name: if the terrain is child of another one, specify the name of the parent terrain

Limit to Coords / Min latitude Max latitude: limit the terrain generation to a defined coordinate range: both values are a range between 0 and 1 which express vertical position relative to whole world length

Generation Algorithm: the algorithm use for terrain, you can select between:

- *FILL_PERCENTAGE*: generate terrain until a certain percentage of available tiles reached, define the following parameters:
    - ***Fill Percentage***
    - ***Cohesiveness***
    - ***Find Fillable Tiles Before Generation***
    - ***Max Failed Fill Iteration Checks***

Last two parameters are used respectively to pre-calculate the fillable tiles (instead of checking every time a fillable one) and limit the check to find a fillable tile, on each iteration: this because the Fill algorithm is prone to deadlocks, depending on the situations.

- *PERLIN_NOISE*: generate terrain using Perlin-Noise algorithm, specify the following parameters:
    - ***Noise Scale Factor***
    - ***Noise sampling threshold***
- *CELLAR_NOISE*: generate terrain using Cellar-Automata algorithm, specify the following parameters:
    - ***Start Chances***: probability terrain will be set for each tile a first iteration
    - ***Passes***: number of iteration passes for the algorithm

Replaceable Terrains: terrains this one can replace; a terrain cannot generate over another already generated terrain (on same level) if not specified here

Avoid Adjacent Terrains: terrains this one will not generate nearby

Terrain names should be unique for all the asset entries. In the same way, since multiple of this asset can be set for the DataGenerator component, terrain names should be unique for all the entries between all assets.

TerrainTilesData

Context menù: Create -> Overworld Generator -> Base Generation Data -> Terrain Tiles Data

This asset defines the terrain *tiles*, it is the world units which comprise the whole overworld. Terrains generated by *TerrainData* are mapped in specific terrain tiles, so this asset can define a default mapping between the two types. Remember the relation between terrains and terrain tiles is *many to one*, meaning that more terrain could be associated with the same terrain tile.

The terrain tile data asset organizes each entry on different *levels*, this is because terrain tiles are organized on different levels, so that on a specific Overworld [x,y] coord can co-exists different terrain tiles, in different levels. The *total levels' count* for each tile is defined at initialization time, based on the biggest level used.

You can define the number of levels for each asset file, like you can move and delete each level with its own content, but currently there's a limit on the possible number of levels.

For each element the following fields are defined:

Terrain Tile Name: name associated to this terrain tiles; terrain tiles' names are not to be confused with terrains' names

Default Associated Terrains: name of the terrains the terrain tile can be associated with; the relation is one to many, since each type of tile can be associated with more terrains.

Use Different Asset Name / Custom Tile Asset Name: allow to specify a different name for the asset to use (tile or sprite), otherwise an asset with the same name of the terrain tile is searched

Use Tile Transitions: if for this type of tile are expected edge-transitions

Set Default Tile Transition Prefix / Default Tile Transition Prefix: allow to specify a suffix-prefix for the edge.transition; if set the prefix will be appended BETWEEN tile base name and specific transition suffix, in the name of the final transition asset to search

Specific Tile Transitions Prefix: suffix-prefix to apply for specific, for each entry specify the used suffix and the list of other terrains for applying it

Avoid Non Standard Transition: if set generator will try to avoid non standard edge-transitions; standard edge transitions are considered single edge borders and corner borders

Allow Isolated Tiles: if Avoid Non Standard Transition is set, will anyway allow tiles with transitions in all 8 directions

Use Custom Transition Suffix Configuration: allow to specify a custom transition configuration, passing the relative data asset

Transition Priority: the transition priority assigned to this terrain tile; this value is used to decide when edge-transitions actually happen: if the priority is greater or equal than the priority of the nearby terrain, with which the transition happens, the transaction actually happens, otherwise no. Generally with which each terrain has a transition depends on the specific tileset

Use Default Tilemap / Default Tilemap Name: by default the terrain tiles will be rendered using Tilemaps' tiles and these will be set on a tilemap with this name (if found in scene)

Default Tilemap Edge-Transition Overrides: specify different tilemaps for specific edge-transitions with other terrains

Use Default Sorting Layer/ Default Sorting Layer: by default the terrain tiles will be rendered using Sprite Renderers and these will be assigned to the specified sorting layer.

Default Sorting Layer Edge-Transition Overrides: specify different sorting layers for specific edge-transitions with other terrains

Just like terrain names, terrain tile names should be unique between all passed terrain tiles data assets.

MapObjectsData

Context menù: Create -> Overworld Generator -> Base Generation Data -> Map Objects Data

This asset defines the map objects to generate in the overworld. Map objects are basically those objects in the world which are considered to remain forever at the same position, like cities, villages or other environmental objects. Map objects can anyway be interactables, like visitable world locations (*villages*, *dungeons*, etc.) which can load another game scene, or objects which allow to collect items (like *chests* or similar), and can have a status which can be permanently changed (again think an opened chest), even if their position is permanent. For each map object instance can also be generated a unique name, again you can think of the names of the different cities or villages.

For each element the following fields are defined:

Map Object Name: name of the map object to generate

Map Object Level: tile level over which map object will be generated; it can actually be greater than max level defined for terrain tiles

Multiple Spawn / Spawn Count: IF map object has to be spawned multiple times and how many times

Need Min Surrounding Land / Min Surrounding Land Value: if the map object need to be placed on a position with a minimum surrounding land, for example to avoid to be placed on a small island out of the mainland

Min Surrounding Land In Percentage / Min Surrounding Land Percentage: specify the Min Surrounding land in percentage of the whole mainland

Is Terrain Required / Required Terrain Name: if the map object need to be placed over the tiles a defined terrain has been generated

Min Distance From Borders: minimum distance from borders of the required terrain

Limit to Coords / Min latitude Max latitude: limit the map object generation to a defined coordinate range: both values are a range between 0 and 1 which express vertical position relative to whole world length

Min Distance From Same Object Type: each spawn will have a minimum distance from any other of the same map object

Min Distance From Any Object Type: each spawn will have a minimum distance from any other map object

Save As Terrain Tile: Terrain will be saved as a single terrain tile, at the spawn position and the defined level, as a particular case of terrain tile (without any edge-transition); this is useful for example for saving map objects with an high spawn count and which don't require to be interactive. Many of the following are disable is this is set.

Terrain Tiles To Avoid: map object will not placed above any of the defined terrains tiles

Terrain Tiles To Clear: any of the specified terrain tiles will be cleared if at the same position of the spawned map object

Occupied Tiles Size: map object size in tiles it occupies, this is used for Terrain Tiles To Avoid and Terrain Tiles To Clear.

Force Placement: map object will be placed anyway, after a certain number of tries, even if requirements are not met

Generate Name For Object / Name List File: generate a name for each single instance of spawned map object, selecting in the name list of the passed file; generated name will be unique between each instance

Template: GameObject template used on each map object spawn generation

Set Sprite Image: Set the sprite image on SpriteRenderer (if any) on the instantiated map object; by default the sprite loaded will have the same name of the map object's specified name

Use Specific Sprite Image / Sprite Name: allow to specify a sprite name to load different than map object's name

Use Different Asset Name: only if map object is saved as terrain tile, allow to specify an asset with a name different than map object's name

Use Default Tilemap / Default Tilemap Name: only if map object is saved as terrain tile, it will be rendered by default using Tilemaps' tiles and these will be set on a tilemap with this name (if found in scene)

Use Default Sorting Layer/ Default Sorting Layer: map objects will be rendered by default using Sprite Renderers and these will be assigned to the specified sorting layer.

Tip: if Force Placement is set, setting a terrain tile name both in the Terrains To Avoid and Terrains To Clear, will at least clear that specific terrain where the map object is placed in the case it cannot be avoided

Exactly like terrains names and terrain tiles names, map objects' names should be unique between all passed map objects data assets.

Extra utility data assets

TilesDictionary

Context menù: Create -> Overworld Generator -> Tiles Naming -> Tiles Dictionary

Set for *OverworldVisualGenerator* component. This asset defines an extra binding between terrain tiles and the effectively used graphic assets. If tiles and sprites have the same base name (so without considering edge transition suffixes) or in map objects data assets the correct asset name to use is specified, this asset is not necessary.

TilesSuffixConfig

Context menù: Create -> Overworld Generator -> Tiles Naming -> Tiles Suffix Config

Set for *OverworldVisualGenerator* component or for an entry of *TerrainTileData* asset; definesa custom suffix naming configuration for all used terrain tiles or specific terrain tiles. For each entry of the list you specify the specific transitions (on which of the 8 nearby tiles) and the specific tile transition suffix. If you use a custom configuration you should specify all the possible configurations.

RandomPositionObjectSpawDataAsset

Context menù: Create -> Overworld Generator -> Extra Data -> Object Random Position Spaw Data

Settable for each spawning element entry in *RandomPositionObjectComponent*, as an alternative for setting all its spawning terrains.

ItemInfoAsset

It represents the base definition of a collectable item. It just defines base item information like item type ID and name and item ID and name, the final game should use it for the effective item/object

ItemLootDataAsset

Context menù: Create -> Overworld Generator -> Extra Data -> Loot Info

Defines and item loot, its is a list of possible items content, which can be collected in different ways, likes by an *CollectPoint* (which is a *MapObject* subclass) or a *CollectDrop* (which isn't)

BattleEnemyGroupAsset

Context menù: Create -> Overworld Generator -> Extra Data -> Battle Group Info

Information about a battle enemy group, this is a specific group of enemies on a battle encounter (caused either by a random battle or an overworld enemy). For each component is specified the name and the min/max possible spawns; using these names the game should then load the effective enemies.

BattleEnemyGroupEnemyBindingAsset

Context menù: Create -> Overworld Generator -> Extra Data -> Battle Group Per Enemy Bind Data

Set for *BattleEnemyGroupEnemyBinder*, each entry define a possible set of battle groups, each with its own chance, which could be used in a battle caused by a specific overworld enemy

BattleEnemyGroupTerrainBindingAsset

Context menù: Create -> Overworld Generator -> Extra Data -> Battle Group Per Terrain Bind Data

Set for *BattleEnemyGroupTerrainBinder*, each entry define a possible set of battle groups, each with its own chance, which could be used in a random battle encounter, on a specific terrain

Overworld Objects

Many different objects can appear in the generated overworld; *map objects* are an example but they are just one type of the possible objects which could spawn, also dynamically. Generally, in an overworld, you can have things like collectables (treasure chests or drops), “living” elements like enemies or maybe npcs (the player character itself is an “alive” element) or other. Some of these objects are certainly interactables, other ones not and could be just for decorative purposes or obstacles.

You can place any kind of GameObjects in the overworld, if you implement your own game logic, but some utility MonoBehaviour classes have been created. The base class of these ones is **OverworldObject**. It exposes some virtual methods and properties for detecting if the object is interactable, by the player, or not, and in the case what to do on the player's interaction. It also exposes the delegate *OnPlayerInteractDelegate*, which allows for a user defined behaviour when this happens.

Following the existing **OverworldObject** subclasses.

Map Object objects

MapObject: the base class for *map objects*, it defines some utility fields to automatically resize any collider attached at runtime, so it could match the set sprite image. This is basically because, by map objects’ generation data setting, you can automatically set the rendering image of the rendering assets, in the SpriteRendering component of the spawned template, so if any collider is attached, it needs to be resized at runtime. It also exposes a method for data initialization: the VisualGenerator, once instantiated the map object’s template, will check for this component and call it passing the overworld map object’s saved data.

Place: a place/location in the world, it’s an interactable element because it’s expected the player can interact with it: in fact, what is expected when a player come in proximity of a place in the map, is that he can access the relative game area, so it exposes fields for detecting if a place area scene (and what/what kind of scene) should be loaded and in which way, then it manages the scene loading itself (using OverworldGeneratorManager exposed methods)

SpawnPoint: a point in the world which can spawn other overworld objects, at its position or in a random range around it, like characters (nemies or npcs) or collectables drops; it could also spawn multiple objects and detect on what terrains they can spawn. Like the *RandomPositionObjectSpawner* component, it keeps track of the spawned objects and detects when these are destroyed, eventually respawning them; it can be used as an alternative this one anyways, if you want a particular element to be spawned at a fixed position or nearby it.

CollectPoint: a point where collectables can be collected, this could represent a chest/treasure in the world, a harvest point or anything similar. It can be configured so that the contents is a specific collectable or a collectable loot, with chances for different possible collectables. It supports persistent data status, so that, once the content is collected, it can't be collected anymore, once the game is saved.

InfoPoint: a simple information point in the world, which just shows the player a message, like a signpost.

Overworld Character objects

Character: the base class for *overworld characters*; since a character is supposed to move in the world (be it controlled by the player or by an IA) this class exposes basic movement methods, as well as some animation utilities. Depending on the configuration, the movement can be controlled by the physics or just by a simple position update over the time, in any case this class also implements a simple terrain check behaviour, for letting the character to only "walk" on certain terrains or avoid to pass over other ones (for example for avoiding to walk over the sea or mountains); a similar result could be achieved for example setting tilemap colliders for some of the tilemaps), but, depending on the used tilesets, this is not always possible.

PlayerCharacter: the player controlled character, this class implements the functionalities for interacting with the other overworld objects.

EnemyCharacter: an enemy character, so a character which could trigger a battle encounter scene, depending on the current game logic, so it exposes some fields for setting if and what/what kind of battle scene should be loaded on contact. It implements a basic IA for simple movement, player character detection and pursuing.

Other Overworld objects

CollectableDrop: a world object which let you collect collectables (a specific one or a loot), just like the *CollectPoint*; the difference with this one is that it is not a map object, so it isn't part of overworld data, it could spawn in a random position and doesn't have a persistent data status.

Other utilities

Tiles Sprite Renamer Window

Open by the same entry in *Windows menù*. This utility helps you to rename sprites assets inside a given texture file, so that they will have the correct suffixes needed for tiles' edge-transitions.

All assets need to begin with the same name (prefix) and have numeric suffixes enumerated in increasing order from 0; this generally happens if you use the *auto-slice* utility of *Unity Sprite Editor.*

Set the Texture containing the sprites to rename, set the current prefix (their common part of the name), the wanted new prefix and the list of the suffixes: starting from prefix '0' (or '00') they will all be renamed.

General Tips

If needed, use multiple generation assets of the same type: in Data Generator you can specify an array of multiple elements for each asset data type, this can help to maintain organized assets; for example you could use multiple map object data assets for different kinds of map objects (villages and cities, environmental elements, treasure chests in the world, etc.).

Avoid excessive constraints and rules for each generated terrain and map objects, especially multi-spawn ones: for each rule or constraint extra calculation and checks need to be performed, for each terrain or single map object spawn, which, beside the increased calculation time required, could also cause some terrain or map objects to not be added at all. Try rather to organize terrains in an optimal hierarchical way and maybe specify a map object multiple times, with different required terrains.

Both for mainland and terrains, choose generation algorithm wisely: for not too extended terrains you should probably leave the Fill algorithm since Perlin Noise could be invasive, while for mainland the Perlin Noise with Falloff is probably the best for more natural-looking mainland borders.

For the terrains visualization, if possible, prefer the use of Tilemaps over SpriteRenderers: they're obviously more performant. The latter ones have the advantage of allowing each tile to have a different layer sorting order, based on the vertical/y position, and could be used for effect like the player going behind certain terrain elements (the trees of a forest for example). Just prefer SpriteRenderers only when really needed.

If possible use graphical tilemap assets which contain all the transition combinations: also if the generator will try to avoid non standard transition, if specified, for tiles which don't support them, they could actually happen anyway causing not-so-good-looking effects; furthermore increasing the correction passes for the data generator will increase the correction probability but will degrade the performances. Tilemaps like RPGMaker "A" files are a good example of full edge-transition ones.

If you have a complex overworld, make use of extra management assets (context menu "Create -> Overworld Generator -> Extra Data"): they can avoid you to manually repeat many settings.

Contact Info

I'd really appreciate it if you leave a review for my work.

If you need support you can contact me to the following email:

andrea.novaga.dev@gmail.com

For more information you can check my Unity Publisher page:

https://assetstore.unity.com/publishers/52808?preview=1