Documentation

Synnaxium Studio
Head office
99 A Boulevard Constantin Descat
59200 Tourcoing
(+33)3.74.09.69.80

SOMMAIRE

About 5

Getting started 6
- Introduction 6
- Palette 6
- First prefabs 6
- Edimap window 7
- The grid 8
- Pivot 11
- Deletion (Ctrl + left click) 12
- Optimization 12

MapGrid: Grid configuration 15
- Grid Orientation 15
- Color 16
- Layer Height, Layer Min & Layer Max 16
- Cell Size 17
- Optimized Cell Size 18
- Unoptimize On Edit 19
- Optimize Automatically 19
- Map Center Offset 19

Prefab configuration 20
- Configuration levels 20
  - Global configuration 20
  - Palette configuration 21
  - Per prefab configuration 22
- Default Map Object 23
  - Map Object Type 23
  - Size 24
  - Positioning 25
  - Selecting 27
  - Palette Sprite 28

Tiling 29
- Introduction 29
- Prefix 30
- Tiles 32
  - Prefab 32

Flip 33
Rotation 33
Weight 33
Create a tiling (AUTO FILL usage) 33
AUTO FILL configuration 35
Flip X et Y 35
Reverse Self 36
Allow Rotation 36
Tiling Type 36
Full 36
Horizontal 36
Vertical 37
Joined tilings 38

The Edimap window 40
Grid 41
Edit 41
Brush 41
Hide Layer Below 41
Layer 41
Size X & Size Y 42
Shape 42
Palette 43
Folder 43
Object grid 43
Misc 44
Show Hotkeys 45
Detect Depth 45
Selection 45
Unoptimize On Edit 45
Optimize Automatically 45
Execute initialization scripts 45
Sort map hierarchy 45
Dangerous actions 45

SceneView 46
Preview 46
Focus, Rotations & Flip 46
Hotkeys 47

Execute a script while editing 48

MapObjectInit : Folder configuration 48
IMapObjectInit : Object configuration 49
AdaptativeSortingLayer 49
RendererOrder 51
Y Factor 51
Pivot Child Name 51
Optimization 52
Prefab design 52
Technical details 54
Usage 55
SpriteRenderer 56
Colliders 2D 56
MeshRenderer 56
BoxCollider 56
Customized grid 57
How it works 57
Coordinates 57
OnWorldToCell 57
OnCellToWorld 58
OnDraw 58
OnCellDelta 59
Example: Hexagonal grid 60

About

Edimap is both a map editor and a level editor. What do you think is the difference?
It's very simple: Edimap manages both the design of the environment and the integration of the gameplay. To do so, Edimap proposes a management entirely centered around the prefab.

At Synnaxium Studio, Edimap is our fundamental production tool, used on a daily basis to accelerate the production of our various games (mainly Serious games).

Edimap is able to manage most of the problems you will encounter when developing games based on a cell system, whether it is a platformer, isometric, 2D or 3D game.

We wish you a good reading!

Meet us on Discord !

https://discord.gg/tHWvyraN3B

Getting started

Introduction

The aim of this part is to allow you to use Edimap quickly without necessarily stopping on the technical details.

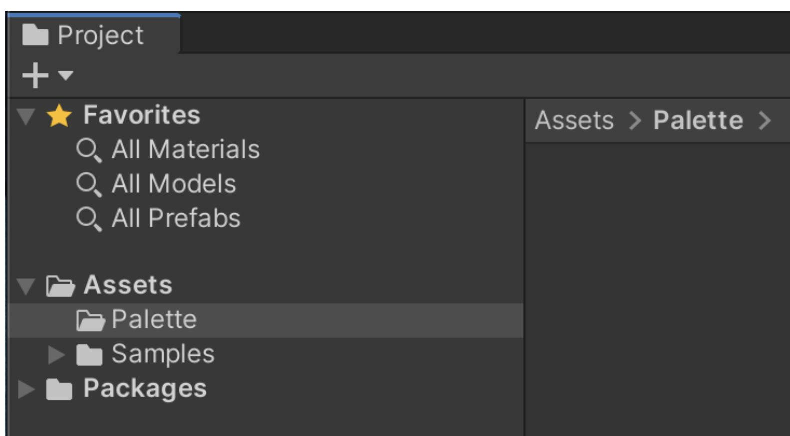
Once you have taken Edimap in hand, the rest of the documentation will explain each feature.

⚠ **If you get an error on "group.Name", consider switching to C# 4.x or higher. The future is now, old man!**

⚠ **If your project is using either URP or HDRP, you will need to change the materials used for the previews (see** Global configuration **section).**

Palette

Let's start by initializing the palette. The palette contains the list of prefabs that you will be able to use during the creation of your map. This palette is materialized by a folder within your project.



First create a folder, where you want it, within your project. You don't have to have this folder at the root of the project.

ℹ Please note that Edimap does not impose a particular path, so you can create other subfolders for organizational reasons.

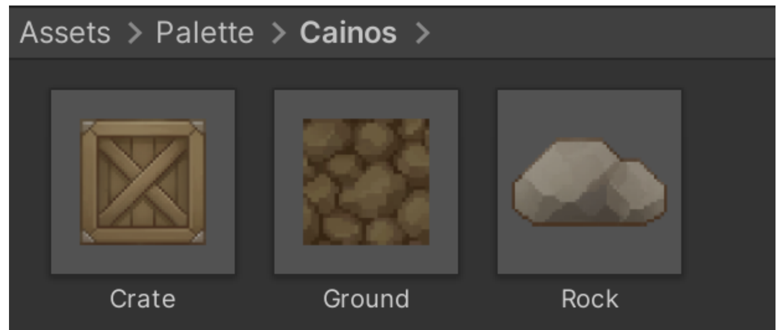
First prefabs

For this part, we will limit ourselves to the simplest use case in order to be able to use the tool quickly, namely 2D games.

Create prefabs with a *SpriteRenderer* component and place them in a subfolder of the palette (in our example the subfolder has the path *Assets/Palette/Cainos*).

⚠ **The subfolder is important, because the palette is made up of prefab folders.**

We thank Cainos for allowing us to use their sprites. They are available in the following folder :



Assets
↳ Synnaxium
↳ Edimap
↳ Examples
↳ Platformer 2D
↳ Graphics
↳ Cainos
↳ Pixel Art Platformer - Village Props
↳ Texture

In our example, we chose the sprites:

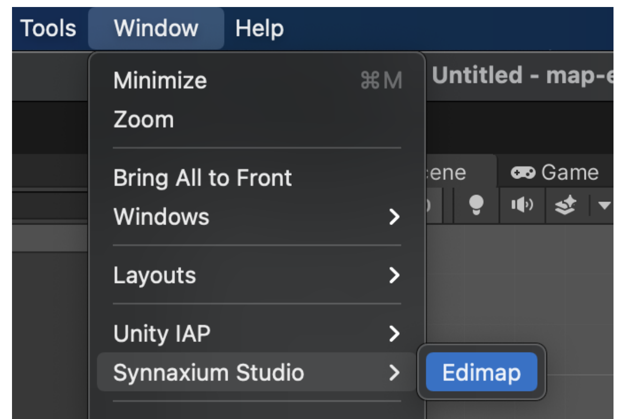
- TX Tileset Ground C TL
- TX Village Props Rock B
- TX Village Props Crate Large

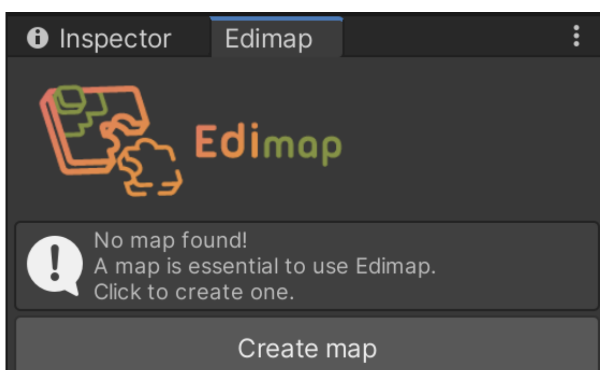
Your first palette is now created. Of course, this first setup is very naive and won't take us very far, but first things first.

Edimap window

Open a new Edimap window using the menu:

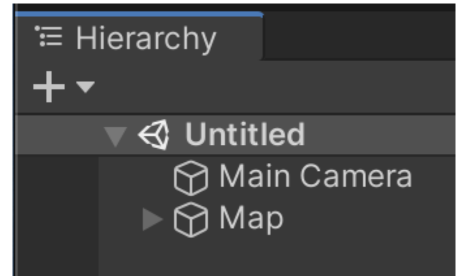
Window ➜ Synnaxium Studio ➜ Edimap





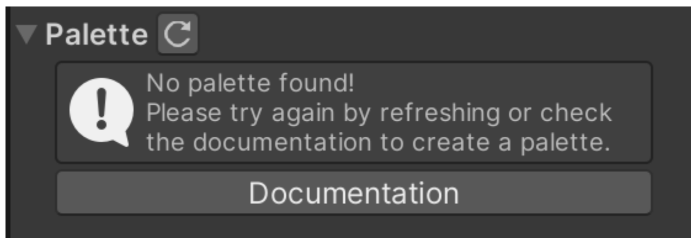
Edimap will appear at the same level as the *Inspector* window, and you will see an information message telling you that no map is present in your scene.

So we start by creating a new one by clicking on the **Create map** button.



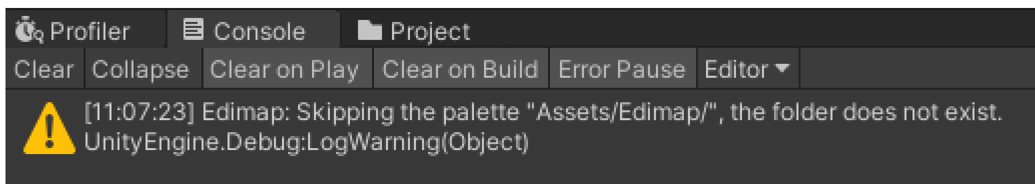
From then on, a *GameObject* named "Map" appears in your scene.

ℹ This is the main object used by Edimap to create a map, the tiles you will add will be included in this object.



By default Edimap uses a predefined path to locate your palette. If there is no palette in the folder in question, an information message will appear in the Edimap window.

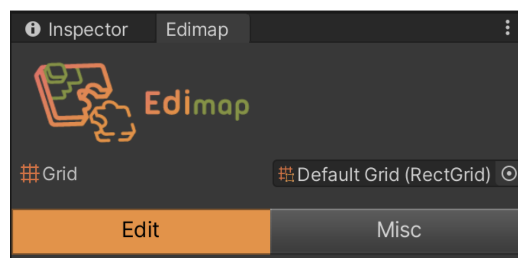
At the same time, a warning should appear in your console:



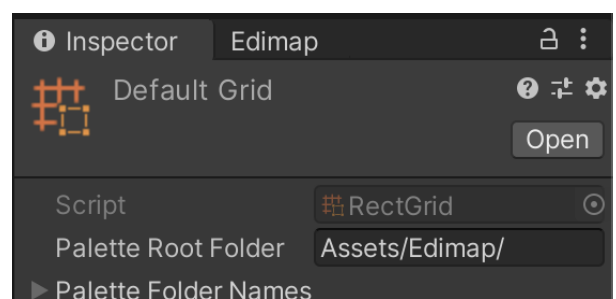
In our example, we prefer to create our own configuration.

The grid

Once the previous step has been completed, you may notice that a reference to a *ScriptableObject* has appeared.



This is the default grid used by Edimap, this one points to the default folder.

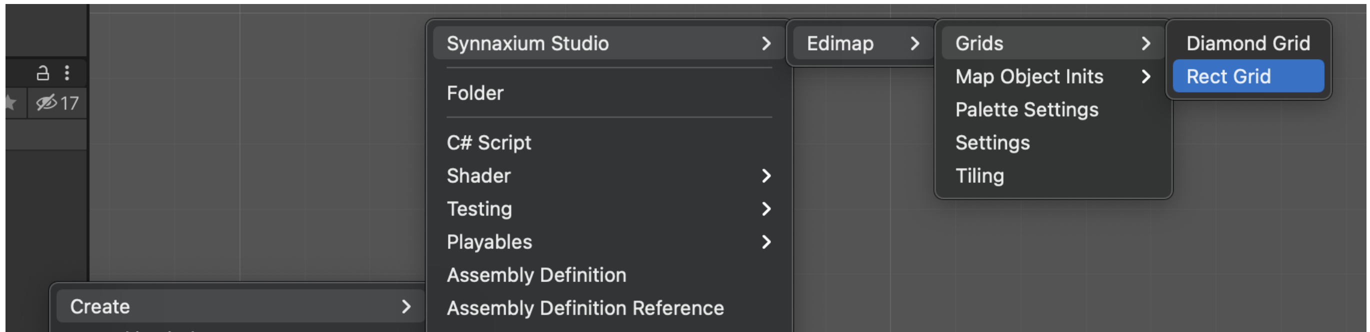


We will replace this grid with a new one created specifically for our example.

For our first grid, we will use a grid to make rectangular shapes.

Start by choosing the folder in which you want to create the grid, then:

Right click ➜ Create ➜ Synnaxium Studio ➜ Edimap ➜ Grids ➜ RectGrid

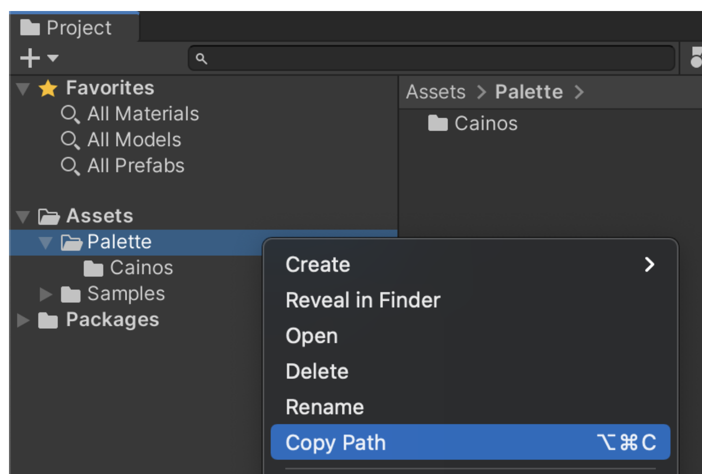


i You can create your grid at the location of your choice in the project.

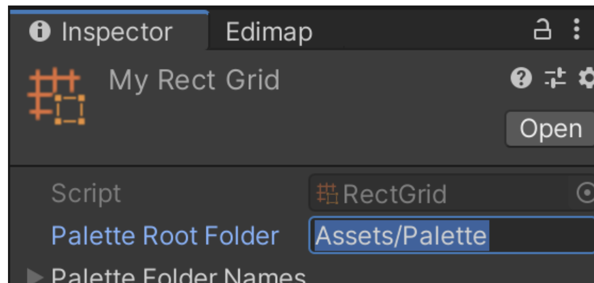
i A grid can be shared between several maps and each grid can point to its own palette. This feature is very handy when it comes to keeping palette sizes reasonable. For example, we could have one palette for the "Desert" biome and another one for the "Forest" biome.

Once our grid has been created we will configure it using its *Inspector*.

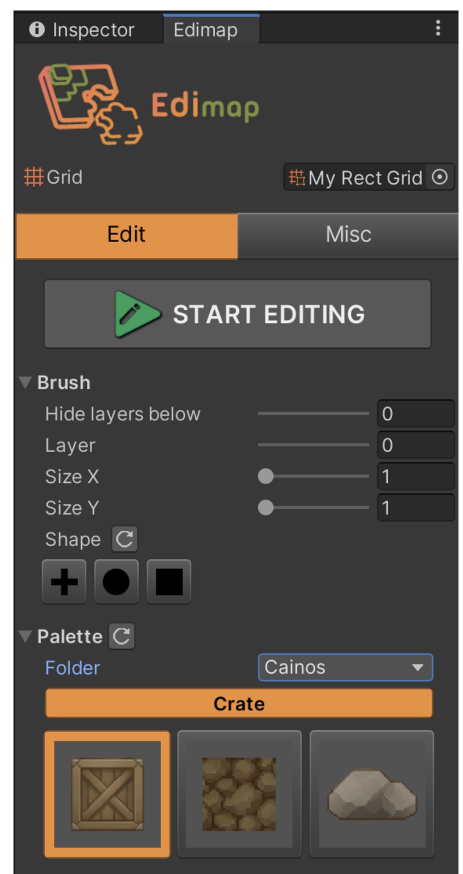
Copy the path to your palette folder (not to be confused with the Cainos subfolder).

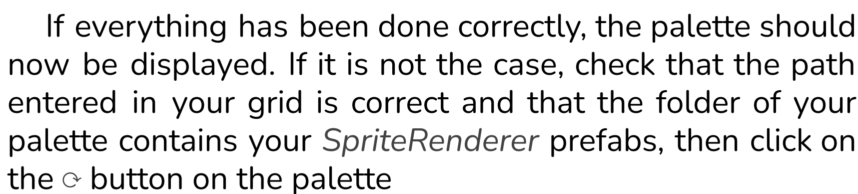


Then paste this path into the **Palette Root Folder** field of your grid.



Now go back to Edimap and select your newly created grid in the **Grid** field.





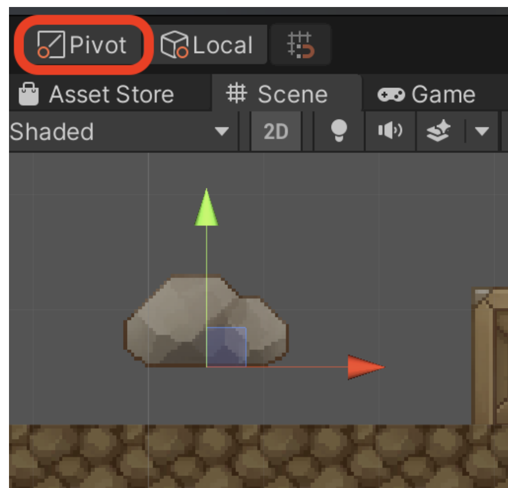
If everything has been done correctly, the palette should now be displayed. If it is not the case, check that the path entered in your grid is correct and that the folder of your palette contains your *SpriteRenderer* prefabs, then click on the ⟳ button on the palette

You are now ready to go.

Click on **START EDITING** and start placing your different prefabs in the scene by left-clicking. Edimap places in the scene the object currently selected in the palette.

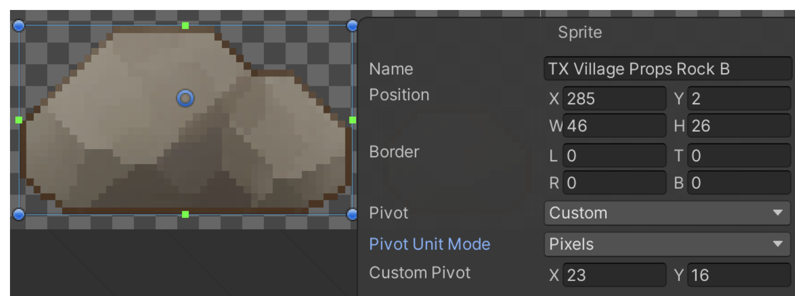
Pivot

The objects created by Edimap are placed in the center of their cell. We must therefore take this into account for the pivot of our sprites.

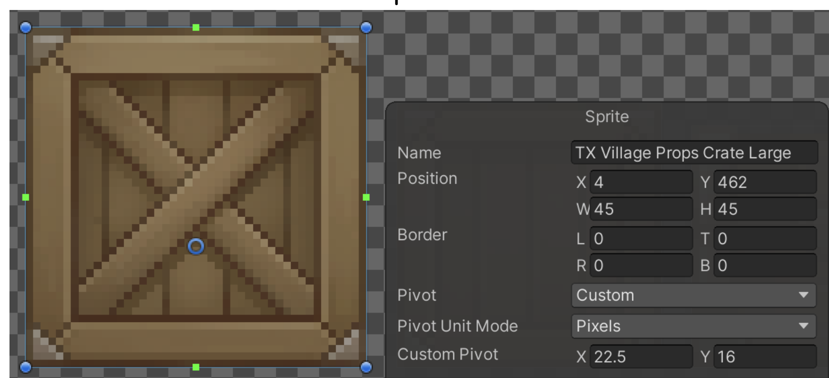


In our case, we notice that the rock is not on the ground. To correct the pivot you just have to :

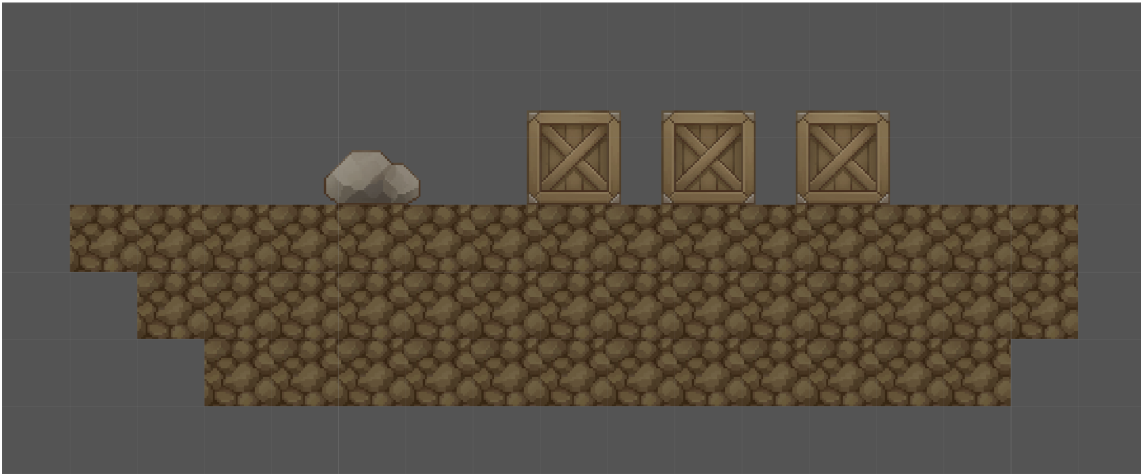
➔ select the sprite of your rock,
➔ edit it by clicking on the **Sprite Editor** button of its *Inspector*,
➔ replace its **Y** value by **16 pixels**, because the cell is 1 unit high by default, and the sprite is set to 32 pixels per unit (16 pixels represents half of the cell).



Make the same correction for the crate sprite.



You will see that the rocks and crates are now correctly positioned on the ground.



Deletion (Ctrl + left click)

To delete an object placed on your map, simply hold down the **Ctrl** key on your keyboard and **left-click** on the object in question, while in edit mode.

ℹ By default, if you create an object in a location where another object of the same type is already present, the oldest one is deleted. To manage overlays, please refer to the Map Object Type section.

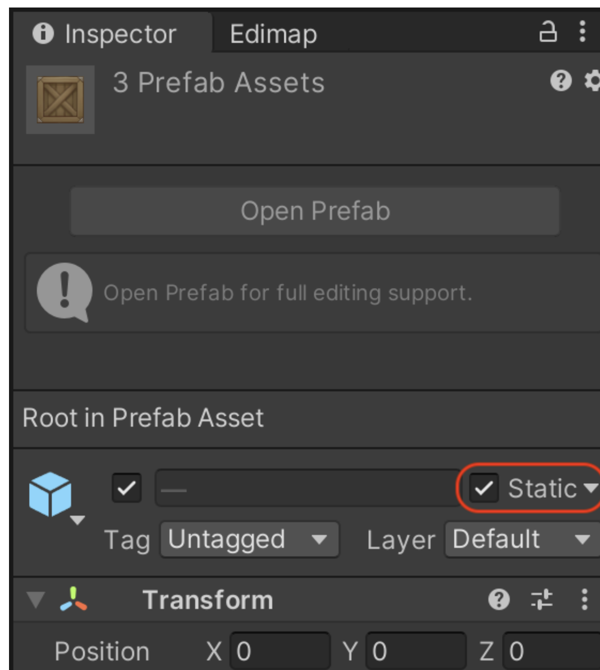
ℹ It is also possible to directly remove a child *GameObject* from the "Map" object from the *SceneView*, however we advise you to be careful when doing so.

⚠ **If the map has been optimized (see next section), it is necessary to "un-optimize" it before you can remove any tiles from it. The** Unoptimize On Edit **and** Optimize Automatically **parameters of your grid are very useful for this.**

Optimization

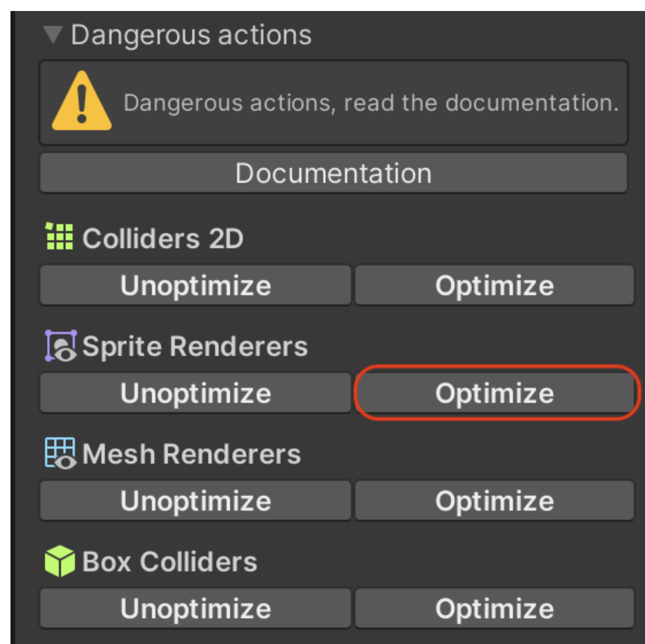
Now that you have succeeded in making your first map with Edimap, we will quickly see how to optimize the rendering of the latter which, as it stands, uses far too much *SpriteRenderer*.

We start by activating the **Static** option in all our prefabs.

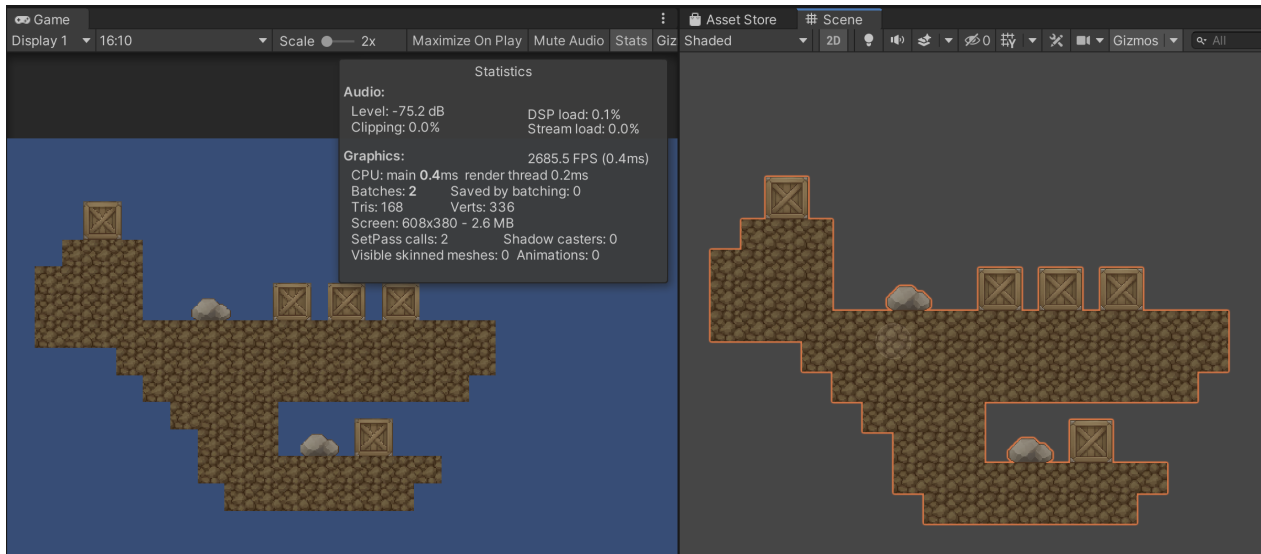


Then we go back to our Edimap window:

- ➔ select the **Misc** tab,
- ➔ unfold the **Dangerous actions** section,
- ➔ Click on the **Optimize** button in the **Sprite Renderers** section.



Your map is now optimized, you can see in **Stats** that now there are only <u>2 batches and 0 dynamic batching</u>.

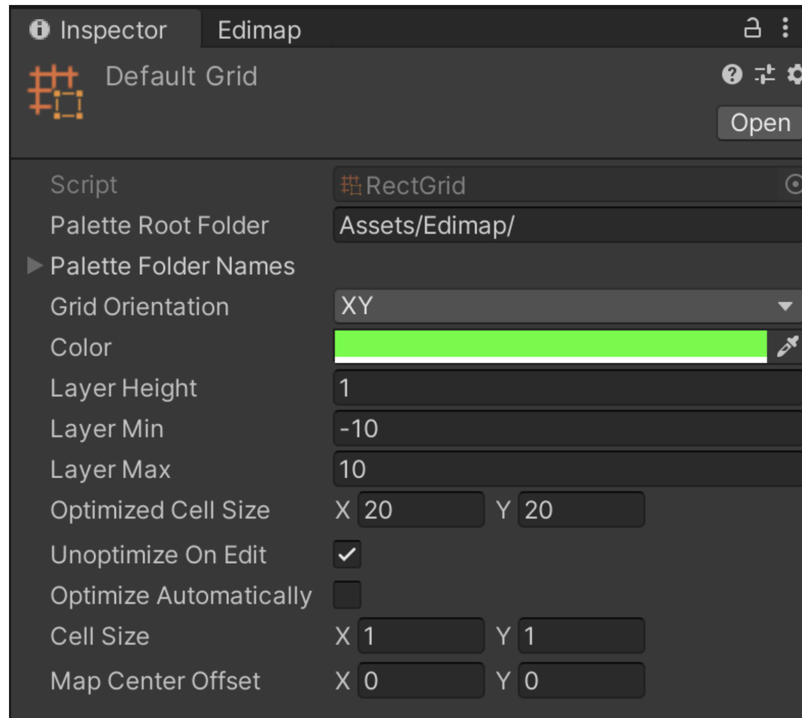


Congratulations, you have just created your first optimized map with Edimap.

However, this one remains very basic, the rest of the documentation will allow you to make more complex maps by presenting you with options that we hope will meet your expectations.

MapGrid: Grid configuration

To better understand map editing with Edimap, we will analyze step by step the *ScriptableObject* **RectGrid**.



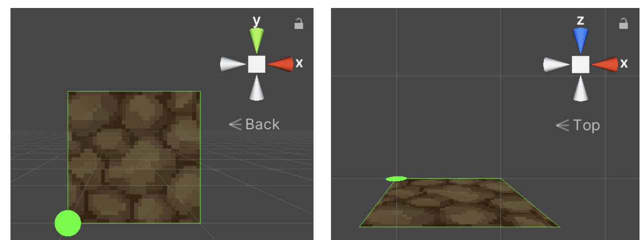
ℹ If you intend to use an isometric grid or a custom grid, we strongly advise you to follow this analysis since the bases are identical between the different types of grids.

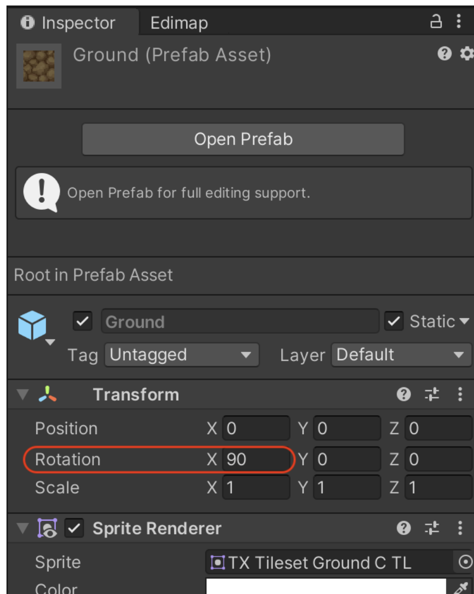
Grid Orientation

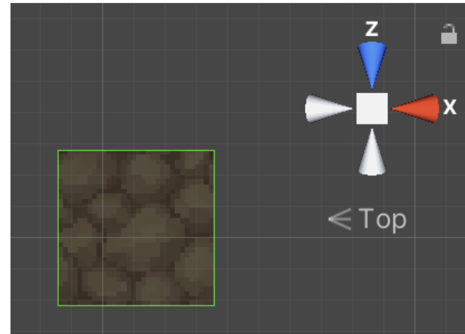
The orientation of the grid follows a coordinate system. Two coordinate systems are available:

- XY ➔ most often associated with 2D maps.
- XZ ➔ generally used for 3D maps.

⚠ **By default, Unity aligns sprites in XY. Therefore, for sprites in XZ orientation, it is necessary to add a 90° rotation in X in the Prefab *Transform* in order to align the rendering correctly.**

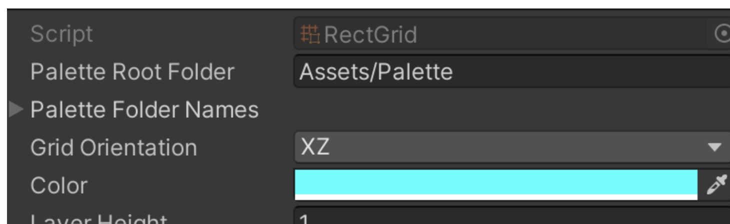


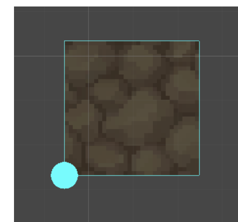




Color

For visibility reasons, it is possible to define a custom color for the grid.





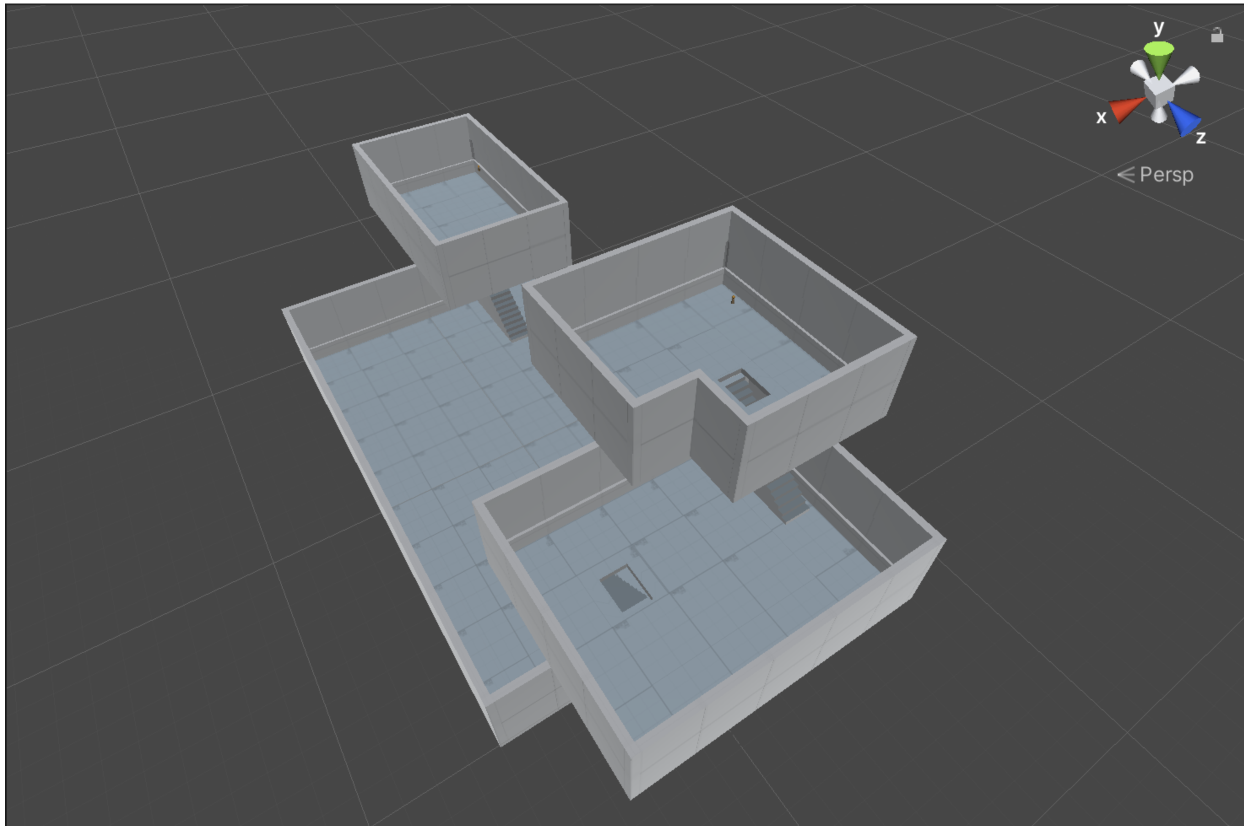
Layer Height, Layer Min & Layer Max

When working with Edimap you can place objects on different depth levels, called "layers".

Each layer has its own depth in the scene space, which is equal to its index multiplied by the **Layer Height** value.

In the case of an XY grid, the depth is located in Z. On the contrary, on an XZ grid, the depth is along the Y axis.

The **Layer Height** parameter can be adjusted on existing maps without any danger.



The depth will be used to build floors for your map, for ground elevations or for buildings.

⚠ **It is important to note that the elevation of decorations such as portraits hanging on a wall should not be resolved by depth. It is preferable to have a prefab that includes this elevation directly.**

There are three parameters for configuring the "layer" within the grid:

- **Layer Height** ➔ defines the distance between two floors.
- **Layer Min** ➔ defines the minimum floor.
- **Layer Max** ➔ defines the maximum floor.

ℹ The **Layer Min** and **Layer Max** fields delimit the number of floors available in a map. They can be modified at will since they are not destructive. Their main interest is to avoid manipulating a slider of 20 values when the map is composed of only 3 floors.

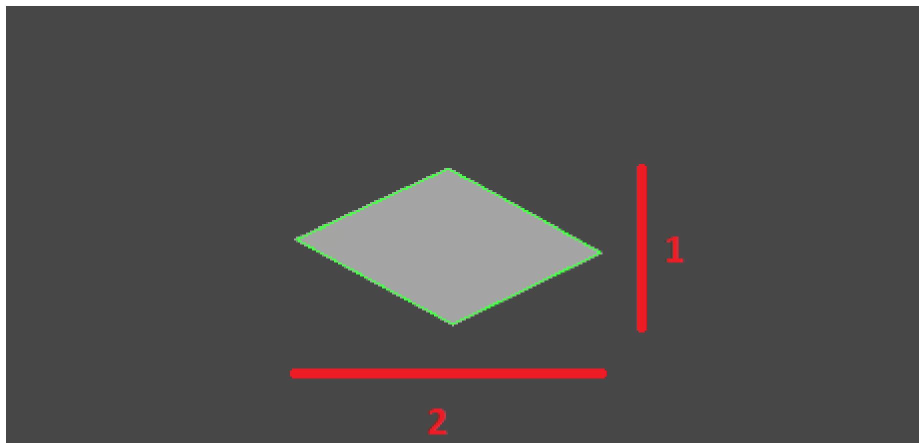
Cell Size

This parameter shapes the geometry of the grid by modifying the size of the cells.

A grid consists of cells and the size of a cell is defined by the **Cell Size** parameter. The default value of 1x1 means that a cell measures 1 unit in the *SceneView*.

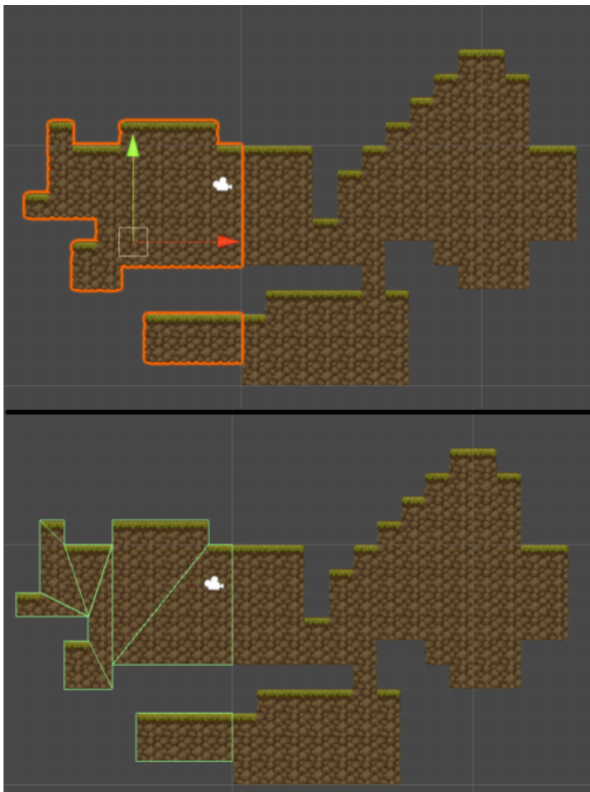
⚠ **Be careful not to confuse the size of the grid cells with the size of an object that can occupy several cells, these are two quite distinct concepts.**

For example, an isometric grid is a grid of rhombuses where each rhombus, i.e. cell, has an X size of 2 and a Y size of 1.



Optimized Cell Size

When Edimap optimizes the renderers and colliders of a map, it is not necessarily wise to merge everything. This parameter is used to define the size of an area to be merged, in terms of number of cells.



In the example opposite, the sprites and colliders are optimized on surfaces of 10x10 cells.

ℹ By default, this parameter is set to 20 in X and Y..

ℹ Ideally, you should choose a size that represents a quarter of a game screen.

Unoptimize On Edit

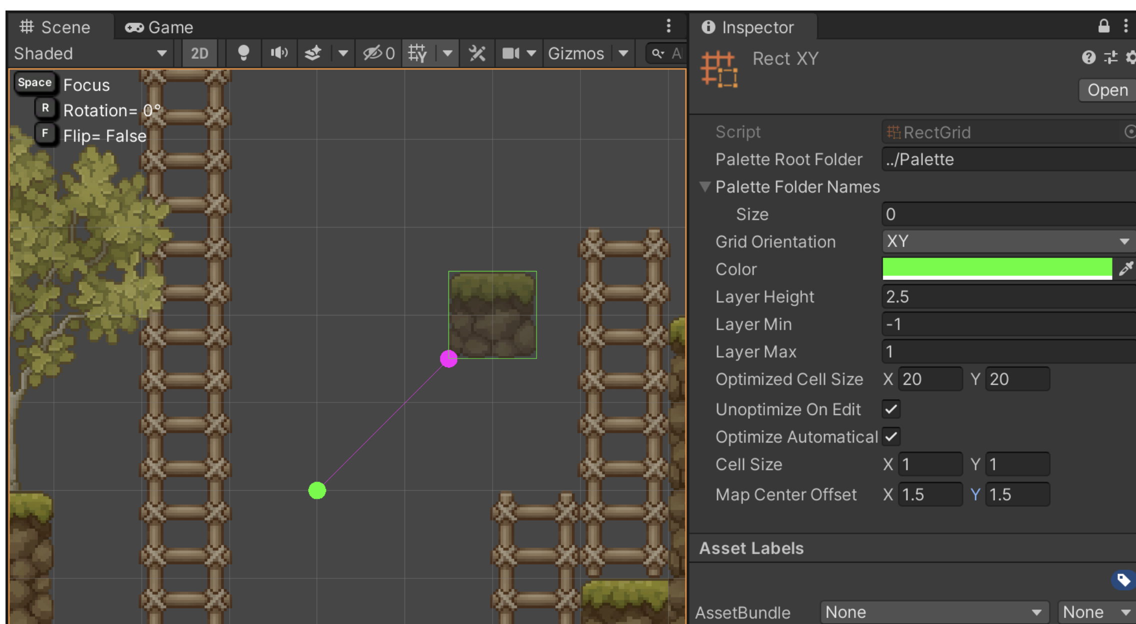
This option allows to remove optimizations from renderers and colliders automatically when editing mode is enabled.

Optimize Automatically

Automatically optimizes renderers and colliders when editing mode is disabled.

Map Center Offset

This parameter is used to arbitrarily shift the origin of the card.



ℹ In edit mode, a point of the grid color represents the origin of the map in relation to the selected layer. The shifted origin is represented by a point of the opposite grid color, and a line is drawn between the two points to be fully aware of the difference..

ℹ By default, the (0, 0) coordinate is the bottom left corner of the (0, 0) cell.

⚠ **This parameter does not allow you to correct the origin of an already existing card, so you should think about choosing the value carefully before you start editing your card. A solution to this problem would be to write a script to shift all the prefabs of the map yourself.**

Prefab configuration

So far, we have only used sprites with the default configuration.

But what if you want to have an object that doesn't fit in the middle of the cell? What if we want to put several objects on the same cell? Or what if we want to place a large house that occupies several cells?

No worries, everything is possible. We will see in detail how to achieve all this.

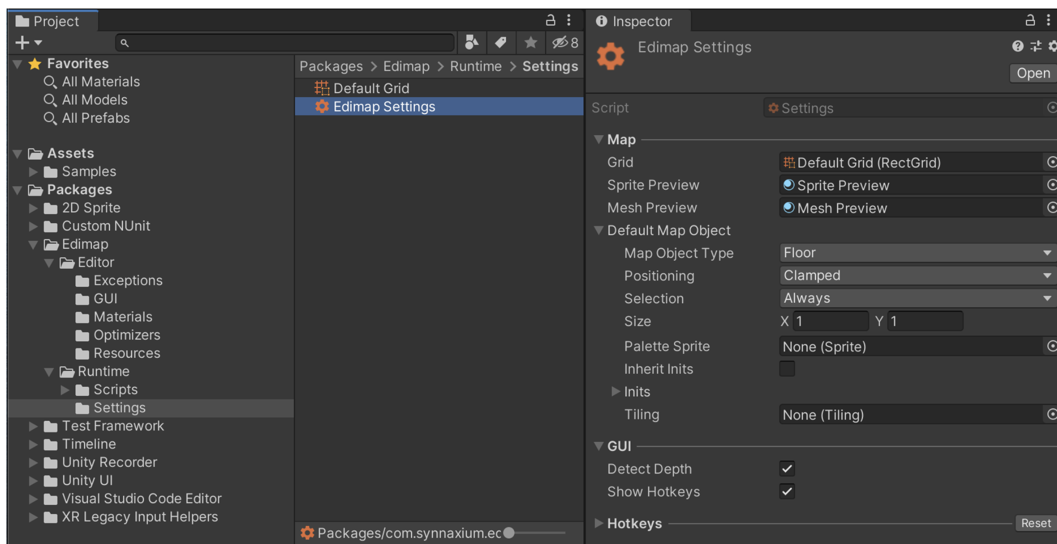
Configuration levels

Edimap allows you to configure your objects in a macro granularity, then by adding finer and finer exceptions.

In case all your objects have the same behavior, you can configure everything at once. If on the contrary your elements are complex and heterogeneous, you will be able to configure Edimap as finely as you need.

Global configuration

Edimap has a global configuration within the package.



ℹ It is possible to create a new instance of this configuration in the Assets folder. This instance will have priority over the configuration in the package folder.

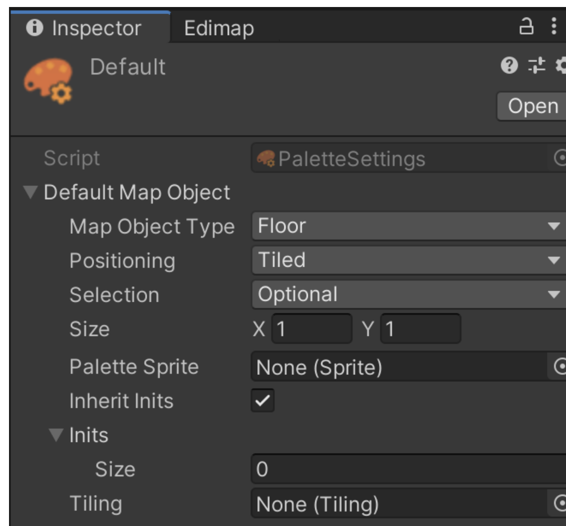
Within this configuration, you will find a **Default Map Object** part which is the default configuration used when creating a prefab within a palette.

This configuration indicates the default values, so it has the lowest priority, but is always present.

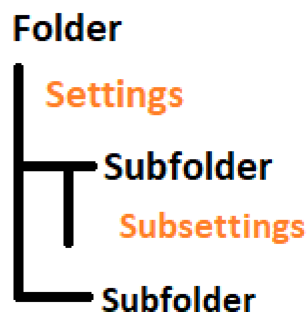
Palette configuration

It is possible to override the global configuration for a folder and its subfolders. To do this, simply add a *ScriptableObject* **Palette Settings** within it.

Right click in the folder ➜ Create ➜ Synnaxium Studio ➜ Edimap ➜ Palette Settings



It is also possible to override the configuration of a palette within one of its subfolders. To do this, simply add a new **Palette Settings** within the subfolder concerned.



So you can have a folder "Props", and a sub-folder "Buildings" which will contain a configuration for large objects.

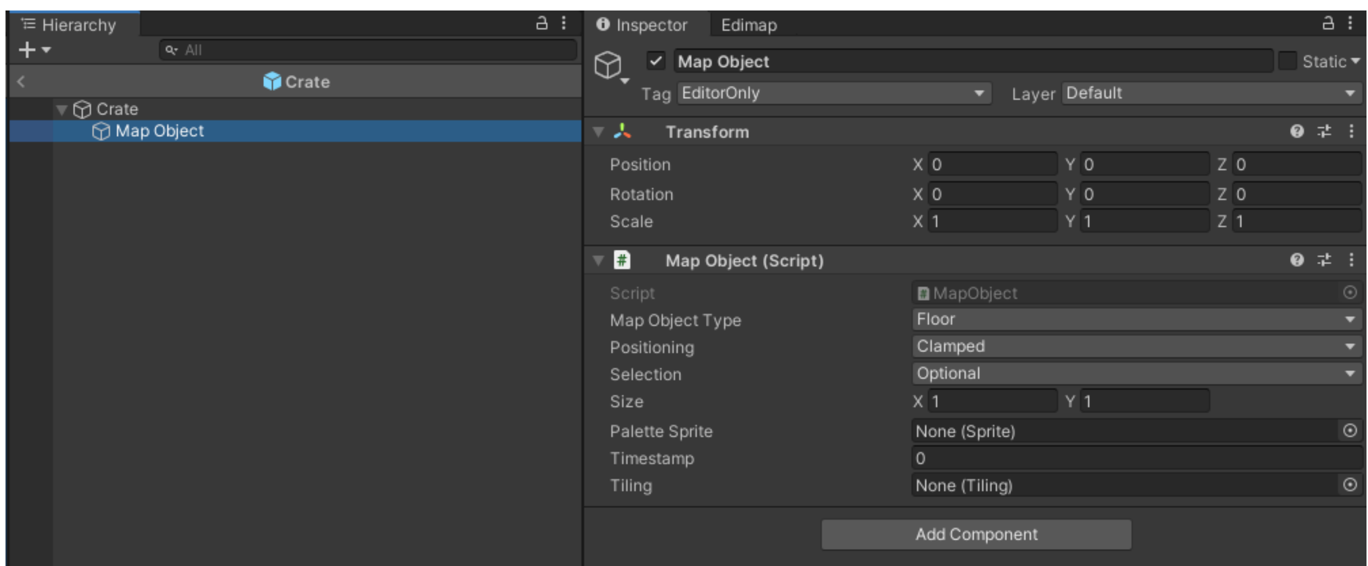
This notion of folder configuration, inherited from the parent folders, allows you to integrate 90% of your objects instantly by simply putting them in the right folders.

Per prefab configuration

Sometimes some objects are too specific, and it can quickly become tedious to have to create an army of subfolders each containing a single prefab.
This is often the case for some very gameplay oriented concepts such as checkpoints and spawns.

Fortunately, it is possible to overload the configuration directly within the prefab:

➔ create a child *GameObject* within the prefab,
➔ set it using the *EditorOnly* tag,
➔ add a **MapObject** component to it.

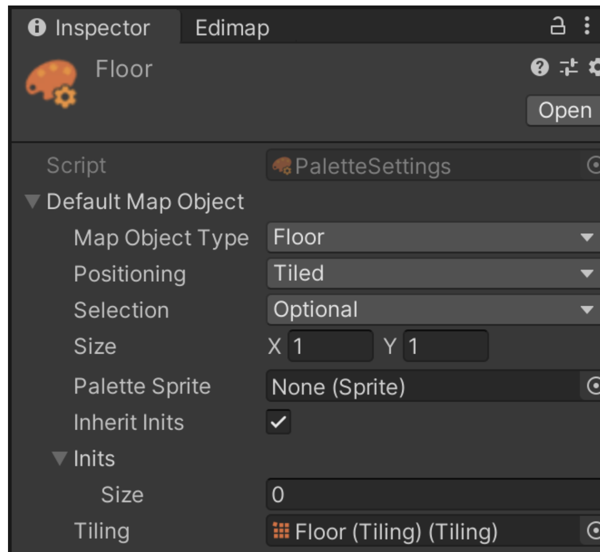


ℹ The *EditorOnly* tag ensures that this approach will have no impact within the executable version.

⚠ **However, there is a limitation to this approach: if the map is instantiated via a prefab at runtime, then the *EditorOnly* tag will not work and the object will be present in the build. This case will occur if you are working on a game containing a procedural generation for a dungeon.**

⚠ **Configuration by the MapObject component always has priority.**

Default Map Object

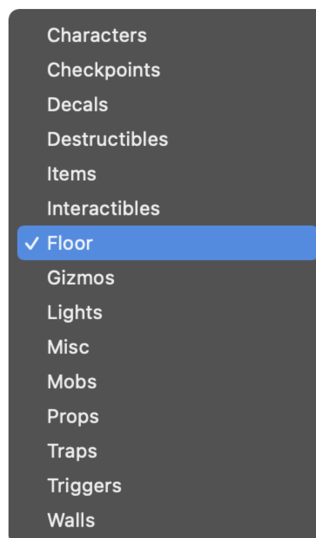


Map Object Type

The type of object is defined by an enumeration named **MapObjectType**.

These types are mainly used to solve overlapping objects. If you put a pebble or a tree on the ground, you don't want the ground to be removed.

The type is also used when deleting manually (ctrl + left click), Edimap removes an object of the same type as the object currently selected in the palette.



i It is strongly advised to use as few types as possible and only one type per folder within the palette to make your life easier when editing a map.

Size

This parameter defines the number of cells occupied by the object.

The size of the object makes it possible to manage overlapping and tiling for floors composed of large tiles. This size is also used when you place several prefabs at the same time.

An army of statues placed at once, without overlapping.

ℹ For performance reasons, the object is actually anchored on its bottom left most cell.

In this case, the new statue will replace the old one because the bottom-left corner of the old one is included in the tile of the new one.

In this other case, the new statue will overlap the old one since it does not touch the bottom-left corner of the old one.

ℹ In practice, this subtlety should not bother you. If despite all this behavior poses real problems within your production, do not hesitate to contact us: Edimap's roadmap remains active!

Positioning

Edimap manages three types of positioning:

- **Normal** ➔ creates the object at the cursor position.
- **Clamped** ➔ positions the object in the middle of the cell.
- **Tiled** ➔ allows you to position the object in the center of a cell while avoiding overlapping.

In the case of objects of size 1x1, the **Clamped** and **Tiled** types will have the same behavior, which will not be the case for larger objects.

In **Clamped** type for 2x2 objects, Edimap allows overlapping.

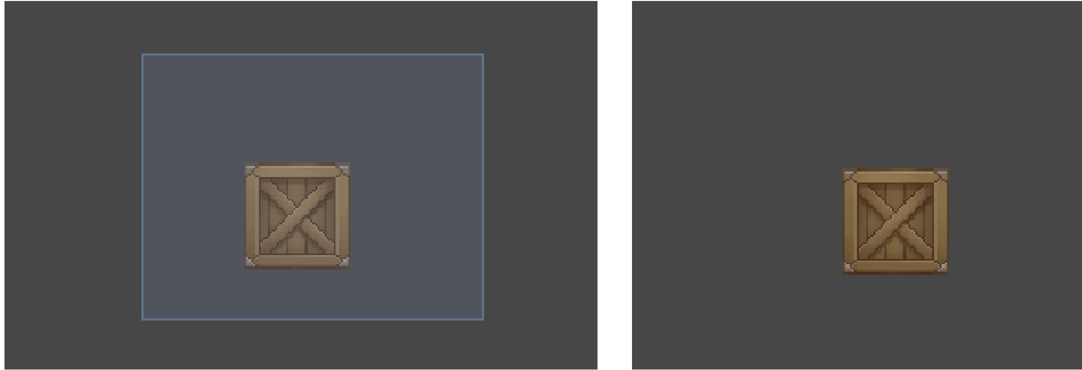
In **Tiled** type for 2x2 objects, Edimap does not allow overlapping.

For the **Normal** type, Edimap considers that your prefabs have the same topology as decals: overlapping is therefore allowed.

The example opposite, shows two rocks which are on the same cell and which are overlapping each other.

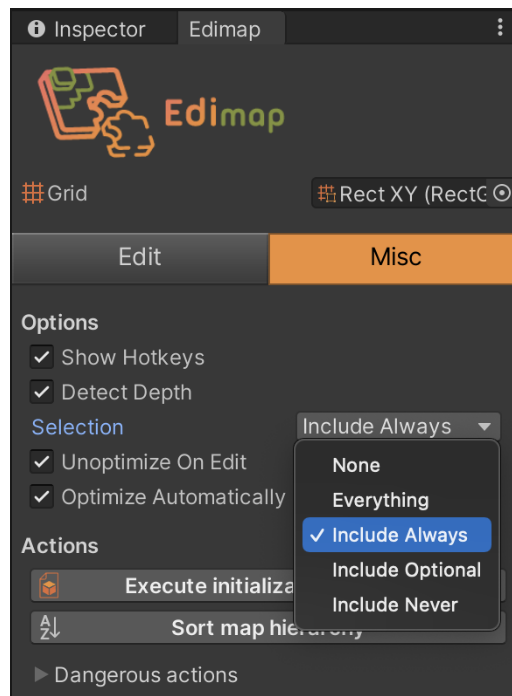
Selecting

During your tests with Edimap, you potentially noticed that you could not select all the objects within the scene.



Nothing happens if you try to select a box.

Edimap proposes a working method by selection. In the **Misc** tab you can notice a Selection option with several possible options.



In addition, each object has one of three selection types:

- **Always** ➔ for gameplay oriented objects often selected by the level designer.
- **Optional** ➔for configurable elements with very little handling.
- **Never** ➔ for purely decorative objects that are part of the decor.

When you create a playable level, you will quickly end up with a cluttered hierarchy and a scene where the selection is imprecise.

By taking care to choose the right type of selection for your prefabs, Edimap makes sure that the level design stage is a real pleasure.

Palette Sprite

Edimap uses *AssetPreview* to get a preview of your prefab. Sometimes, the image generated by it does not necessarily make sense for abstract concepts such as checkpoints.

This parameter is used to add an image for viewing the prefab within the Edimap palette.

Tiling

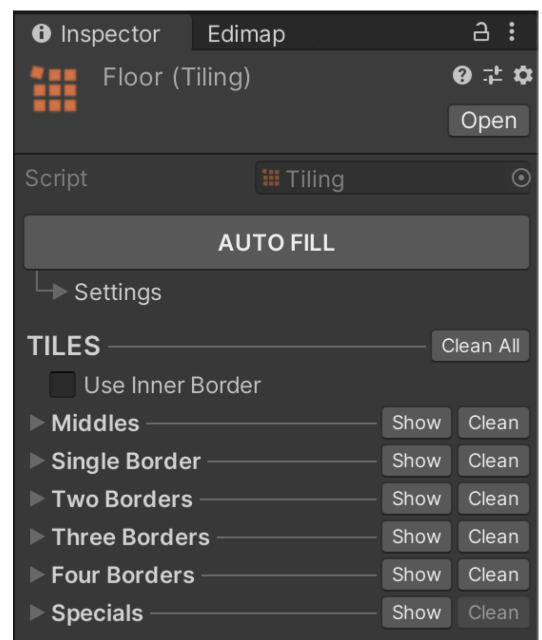
Introduction

When working with certain types of tile, such as walls, ladders, fences or land, we need tiling to create a repeated pattern.

Examples of tiling for a 2D platformer.

To configure a tiling, we use a *ScriptableObject* **Tiling**.

Right click ➜ Create ➜ Synnaxium Studio ➜ Edimap ➜ Tiling



As you can see the tiles are separated into several groups according to the number of borders they contain. There is also a **Specials** group but we will come back to this later.

A large **AUTO FILL** button is also available. It will allow us to automatically fill the tiling with the right tiles according to a certain codification made possible by the use of numerical prefixes.

Prefix

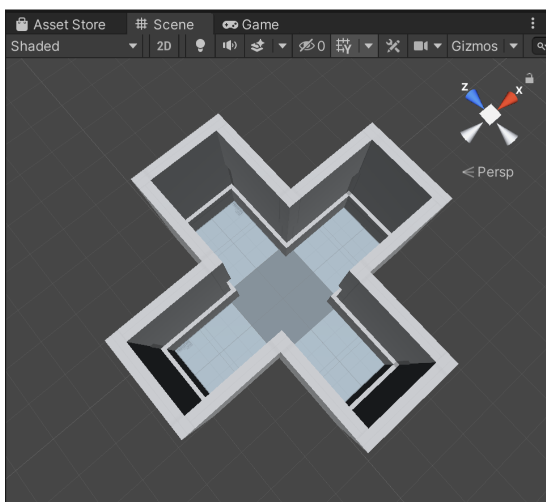
In order to be able to use the **AUTO FILL** feature, it is necessary to respect a standardization based on the numeric keypad of a keyboard.

This standardization seems to us to be the most logical and simple.

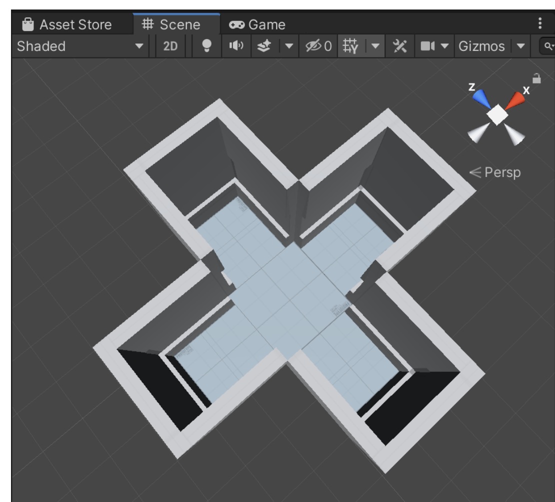
1	bottom-left corner
2	bottom border
3	bottom-right corner
4	left border
5	no border nor corner
6	right border
7	top-right corner
8	top border
9	top-left corner

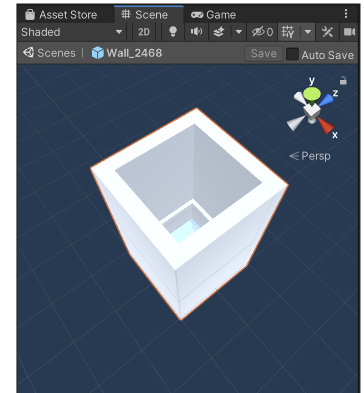
Corners are only used if the **Use Inner Border** option is enabled. The latter is most often used in 3D, especially when a border is represented by a wall.

Use Inner Border *enabled.*

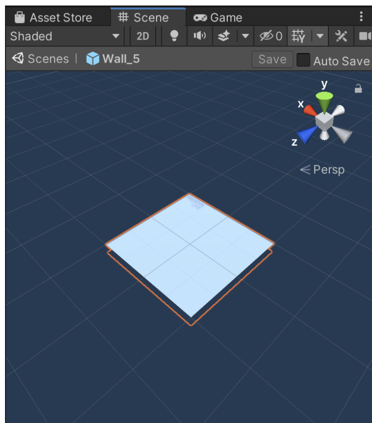


Use Inner Border *disabled.*





A prefix always starts with "_" and can consist of several numbers, so a tile with a name containing the prefix "_2468" will be considered as a tile with 4 borders.

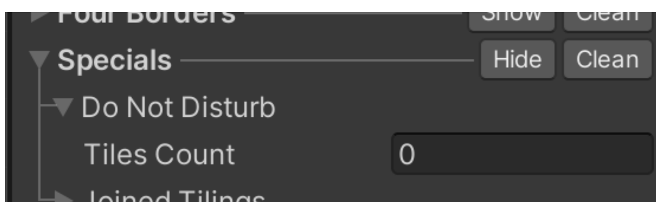


On the other hand, a tile with the prefix "_5" in its name is considered to be a tile without any border, i.e. the tile that will be in the center of 4 other tiles.

The numbers representing the prefixes, as well as a diagram and a mini description, are visible in each group.

Among these groups is the **Specials** group, in which the **Do Not Disturb** subgroup is found. The latter must be filled in directly by hand by the user.

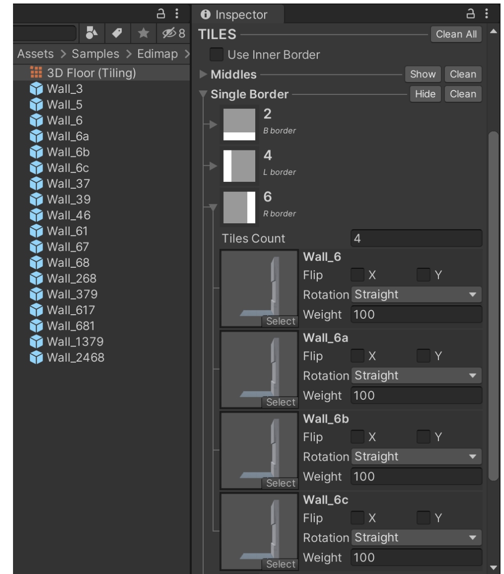
The tiles filled in this table will be considered as neighbors of the tiling and will not be modified by it. This can be very useful to add exceptions to the tiling, such as a gate for example.



As you may have already noticed, it is possible to have several tiles of the same type, i.e. several tiles with a straight border for example (with the prefix "_6"). However, if all these tiles have exactly the same prefix, only one will be taken into account by **AUTO FILL**.

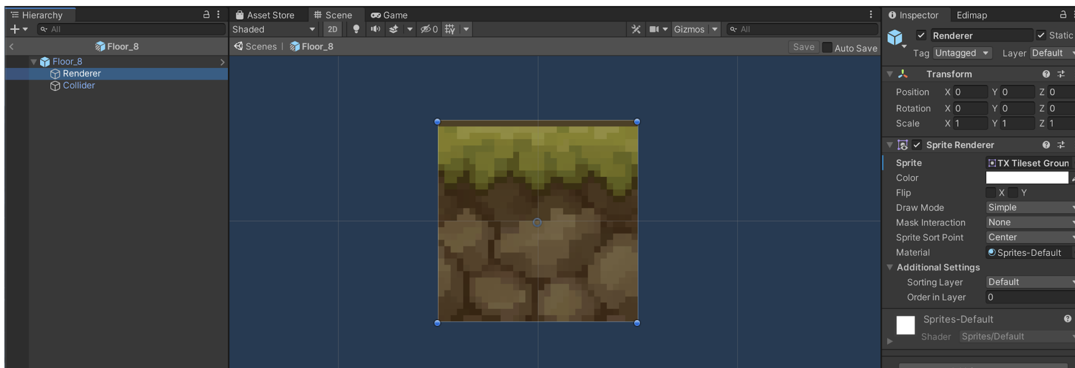
In order for each tile to be taken into account you will just have to add a lowercase letter, which will give such tile names :

➔ *TileName_**6***
➔ *TileName_**6a***
➔ *TileName_**6b***
➔ *TileName_**6c***
➔ *etc.*



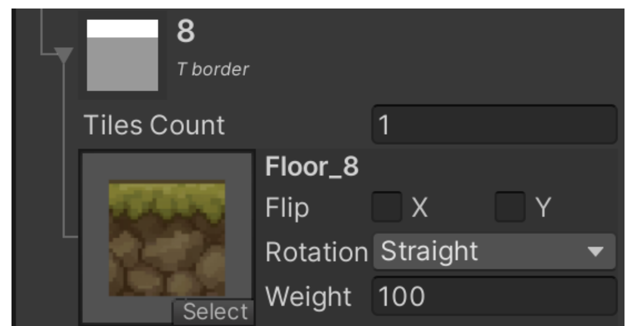
Tiles

A tile is nothing more and nothing less than a simple reference to a prefab.



In addition to the reference to the prefab, the tiles used by tiling have several parameters.

The **Prefab**, **Flip X**, **Flip Y** and **Rotation** parameters are usually automatically completed by **AUTO FILL**.



Prefab

Reference to the prefab used by the tile. The name of the prefab and a preview are visible when this parameter is set

ℹ The rendering of the preview may not be relevant because of *AssetPreview*.

Flip

This parameter indicates whether the tile is inverted on the horizontal (X) or vertical (Y) axis.

Rotation

Rotation of the tile in relation to the pivot point.

Weight

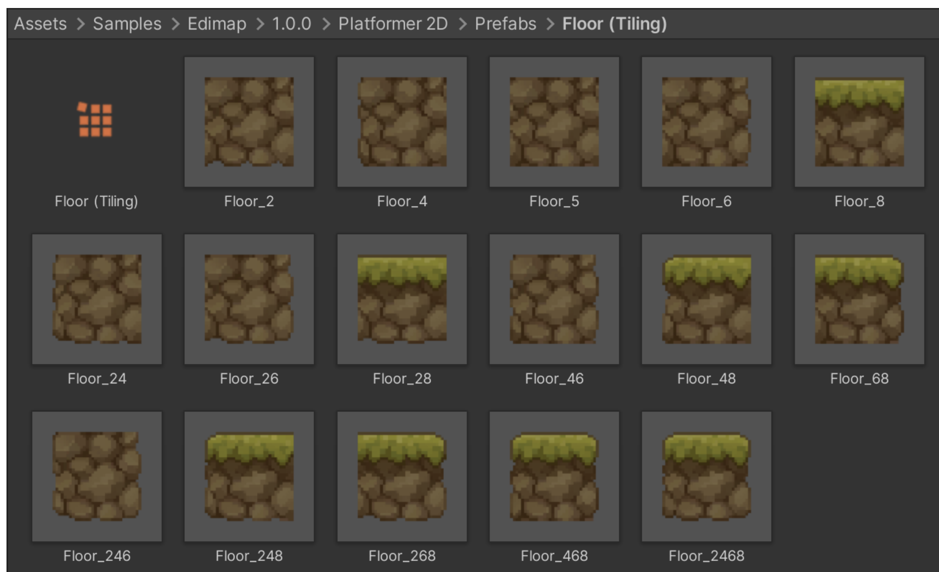
When you have several possible tiles for the same configuration, Edimap will randomly choose one of the possible tiles, weighting the chances by the value of this field. If a tile has a value of 200 and another tile has a value of 100, they will have respectively 66% and 33% chance to be selected.

ℹ The default value is 100 for all tiles.

Create a tiling (**AUTO FILL** usage)

To create a tiling, the simplest method is to prepare a prefab directory containing all the variations.

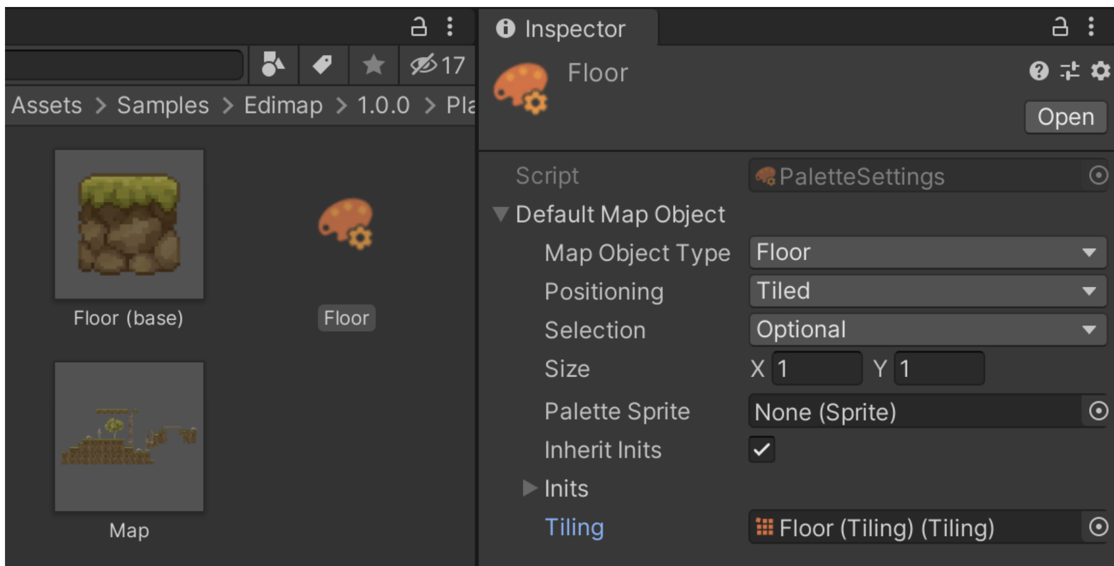
Each variation must be correctly normalized using a prefix as indicated in the Prefix section.



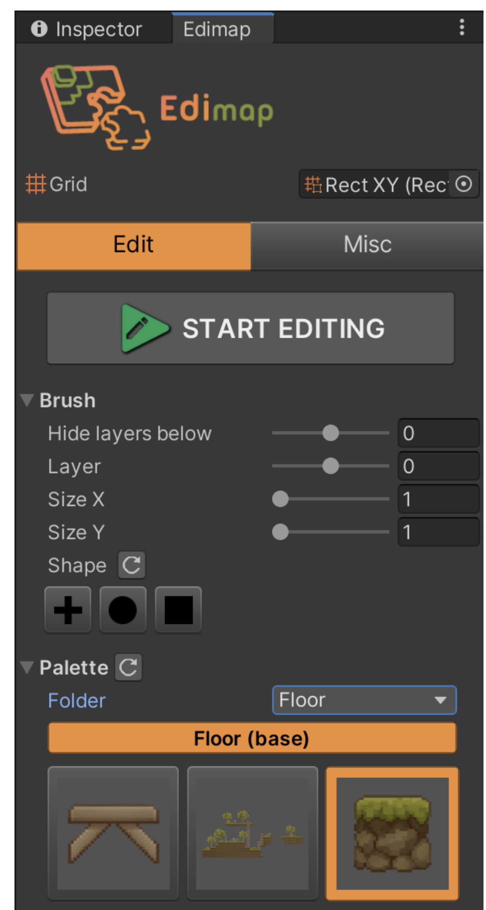
After creating and naming your prefabs, add a *ScriptableObject* **Tiling** within the folder, and click on the **AUTO FILL** button. You will be asked to select the directory where your prefabs are located (the default path is the **Tiling** asset path). This manipulation allows you to automatically pre-fill your **Tiling**.

For the tiling to be operational it must be referenced on a palette:

➔ Create a new folder in the palette folder of your choice.
➔ Add a prefab that will be used only to be displayed in the palette.
➔ Also add a *ScriptableObject* **Palette Settings**.
➔ Select your tiling in the **Tiling** field of this *ScriptableObject*.

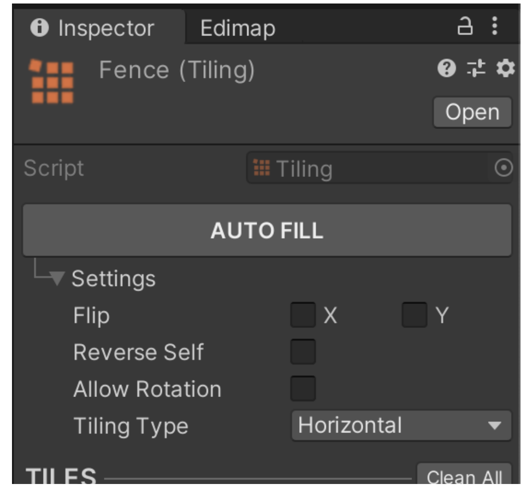


If you proceeded correctly you will see your prefab in the corresponding palette. All you have to do now is test your tiling to check if everything has been configured correctly.



ℹ Two tiling of the same **Map Object Type** can be placed side by side within the map.

AUTO FILL configuration

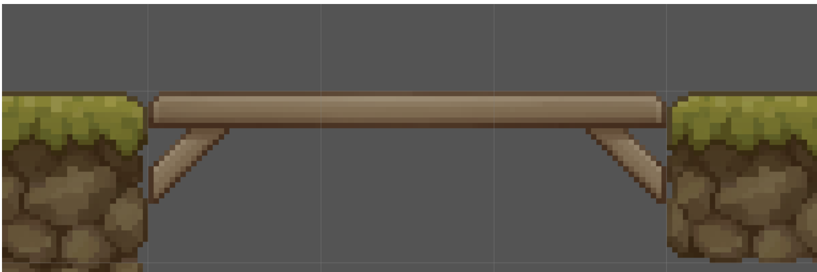


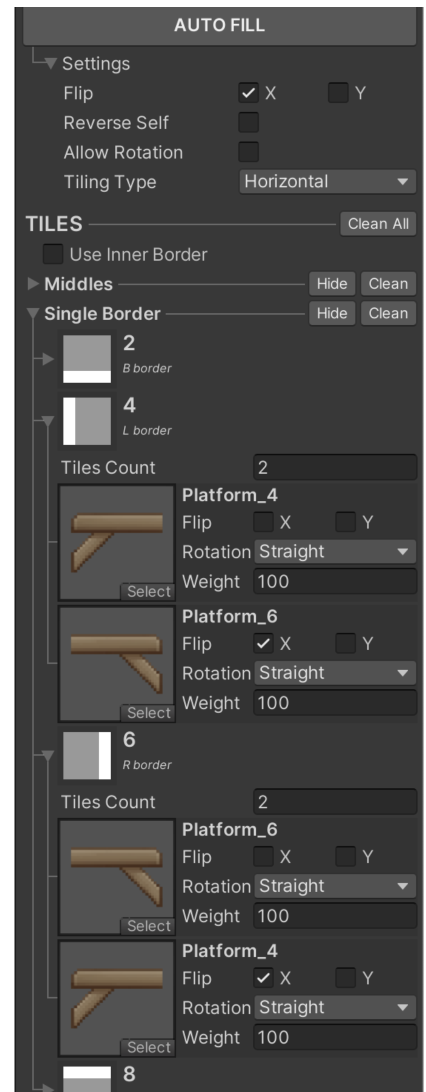
You will notice that there is a drop-down menu named **Settings** under the **AUTO FILL** button of the *ScriptableObject* **Tiling**. These are the settings used for the auto-fill feature.

i By default, the **Settings** drop-down menu is deployed. Just click on **Settings** to hide or show its content.

Flip X et Y

When you make a tiling, it is possible that your left edge prefab also works for the right edge, once flipped.





If the **Flip X** box is checked, then **AUTO FILL** includes horizontal inversions to fill the tiling.

So in the example on the left we notice that the tile with the prefix _4 appears in tile category 6, and its **Flip X** parameter is activated. We can also observe the opposite, tile _6 in tile category 4.

The operation is the same for the **Flip Y** parameter but the inversions are vertical.

i The real interest of this option is to have only one prefab to complete two categories of tile in the tiling.

Reverse Self

This parameter is activated when the same prefab can be present more than once for the same configuration by inverting itself.

ℹ It is mainly used for "corridor" prefabs, i.e. prefabs with only two borders.

Allow Rotation

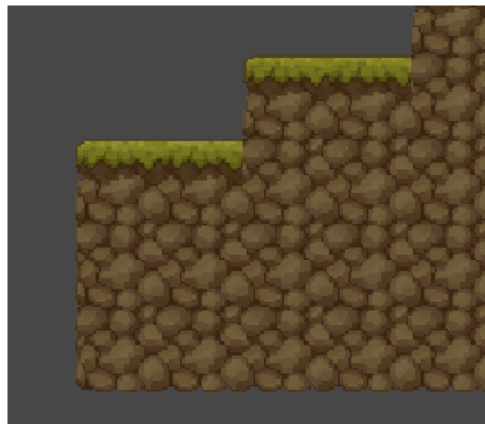
It works like the **Flip** but with rotations. As a result, a single tile prefab can complete categories 2, 4, 6 and 8. The same prefab will be in each category but with a different **Rotation**.

Tiling Type

A tiling type is required for **AUTO FILL** to be able to determine which tiles to add automatically. There are 3 types of tiling:

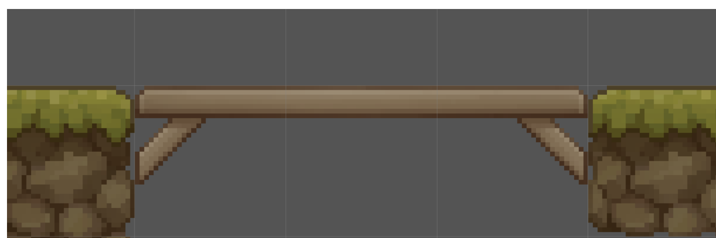
Full

It is used when your pattern repeats itself both vertically and horizontally, which is often the case for the floor.



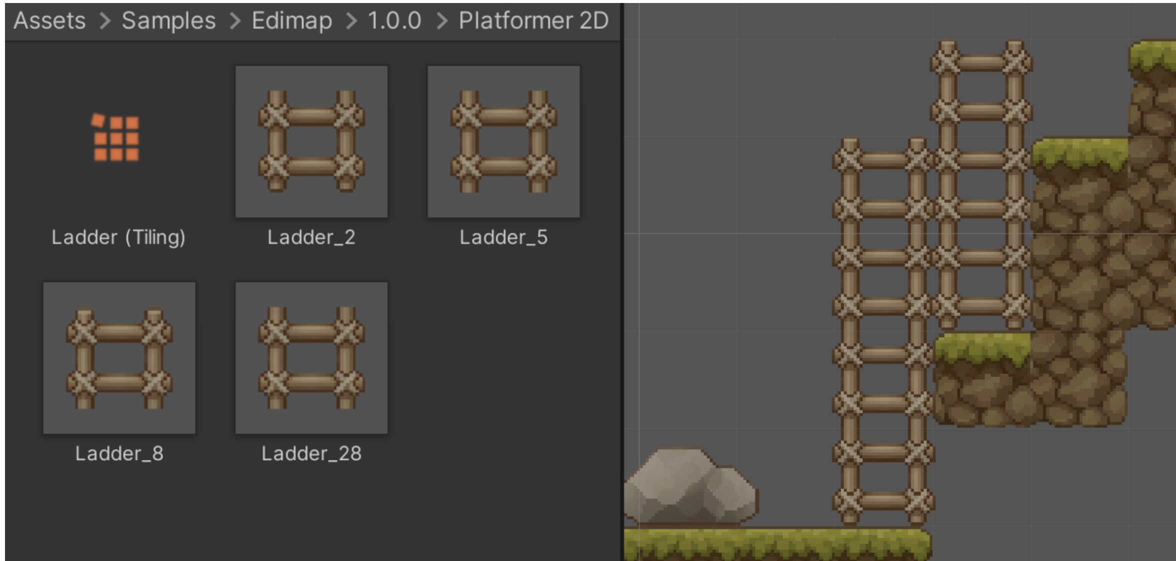
Horizontal

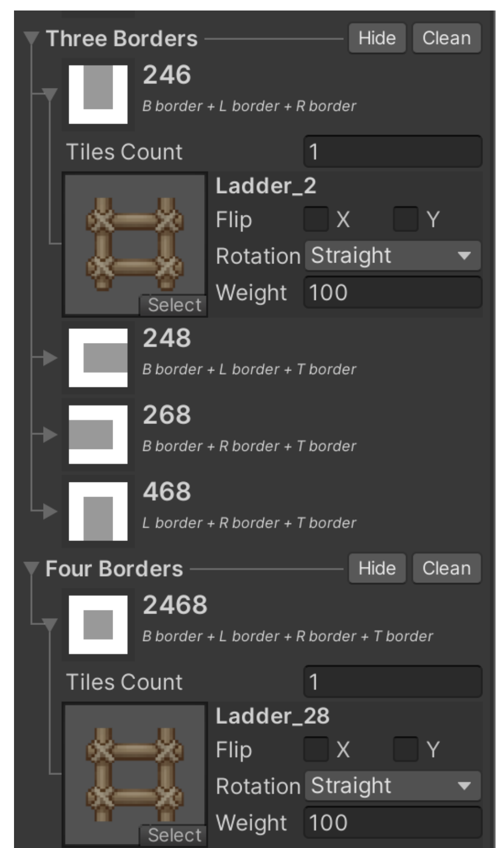
This type is used for horizontal repetitions, in our 2D example we use it for platforms.



Vertical

It is associated with vertical repetitions, such as ladders or ropes.





As previously mentioned, **AUTO FILL** will complete the missing tiles of certain categories, although the prefabs used must be correctly named.

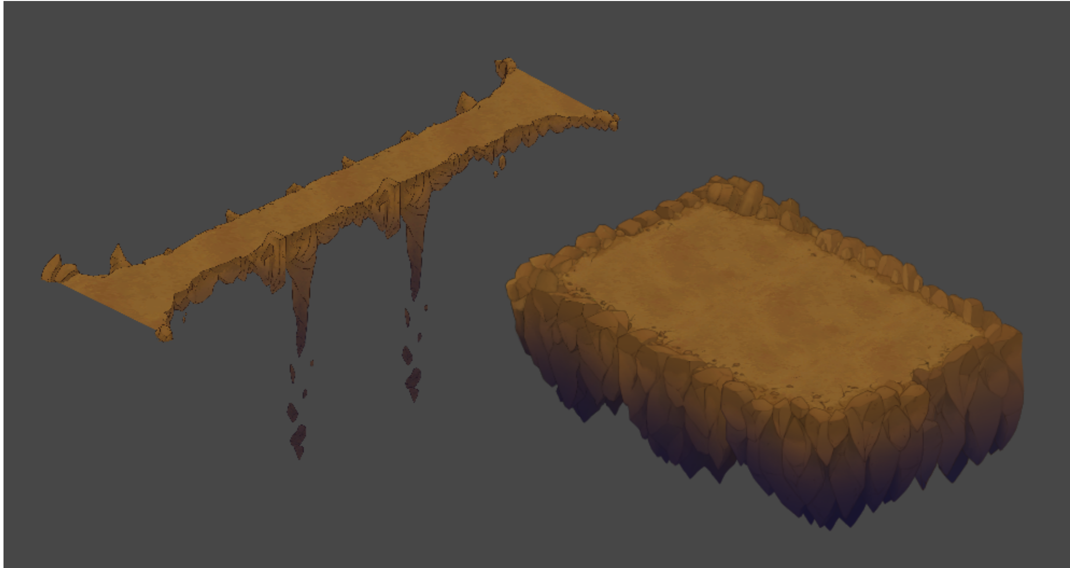
As an example, in our case we only need a bottom border (**2**), a top border (**8**), a tile for the middle (**5**) and optionally a tile for a ladder of a single cell (**28**).

In this way the missing configurations will be completed by **AUTO FILL**.

Joined tilings

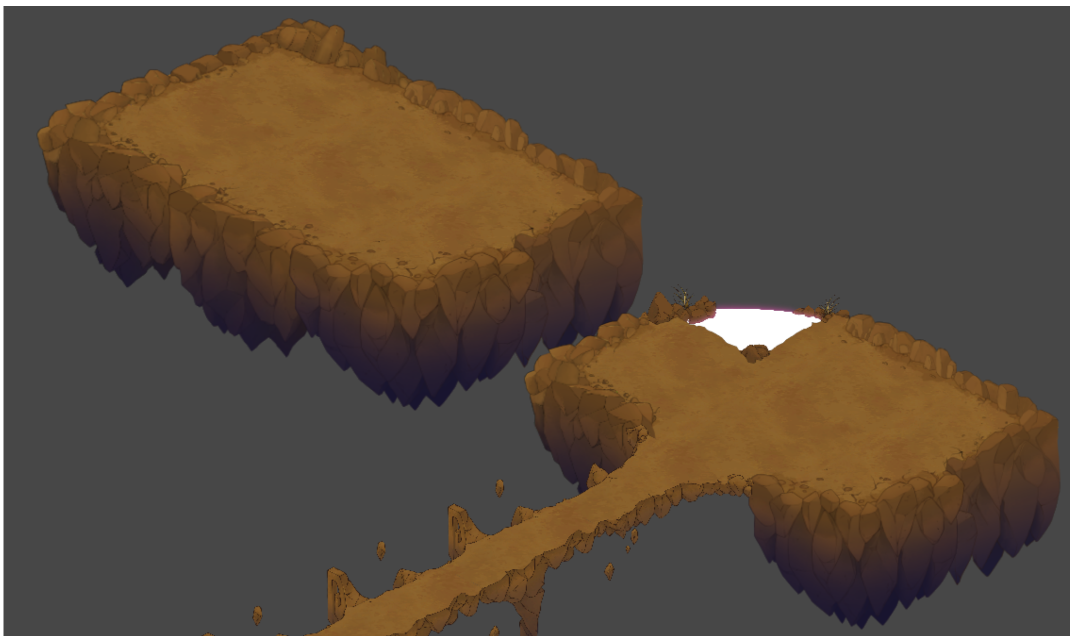
By default, tilings within Edimap do not interact with each other. In other words, if two types of tilings are side by side, each one will close its respective edges.

In some cases, this behaviour is inappropriate. Let's consider this example with two tilings :



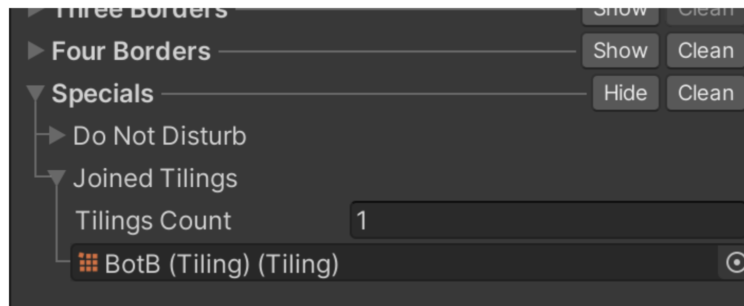
Horizontal bridge tiling and full floor tiling.

The bridge has been designed to naturally extend the floor area, without seam:



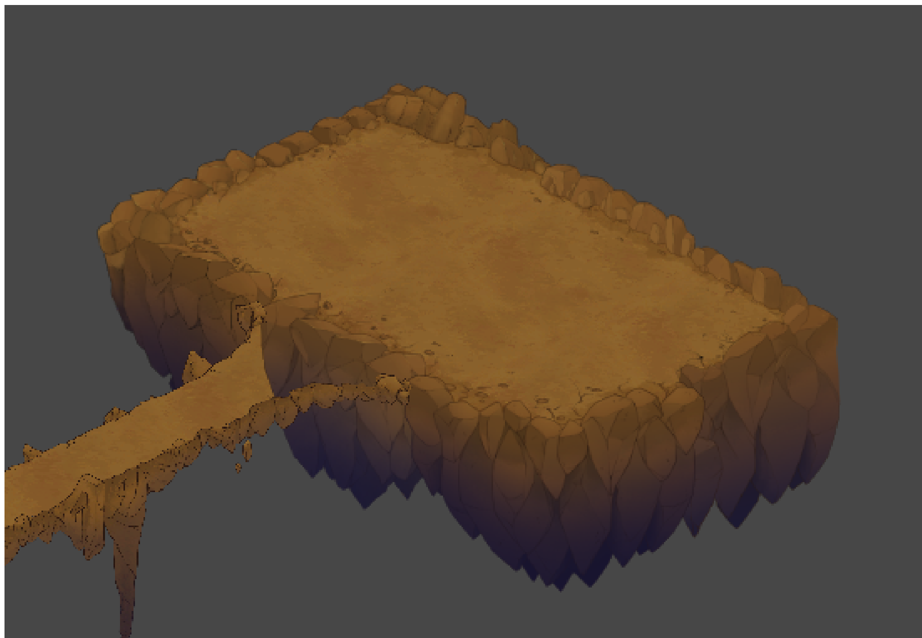
The bridge and the floor interlock to create a homogeneous whole..

In this case, the floor tiling contains a "joint" to the bridge tiling, so the floor tiling does not assign a tile with an edge when it is in contact with a tile of the bridge tiling. The bridge is perfectly independent and places its edge despite the presence of the floor tile next to it.



The ground tiling has a reference to the bridge tiling in the **Joined Tilings** *field.*

Without this option set, the rendering would have been as follows:



Without joints, the tiling of the floor will create a border that is not desired here.

i The joint is a very powerful tool that will allow you to create interior walls, roads, bridges, and a whole bunch of complex repeated patterns.

⚠ **If a tiling A includes a reference to tiling B:**

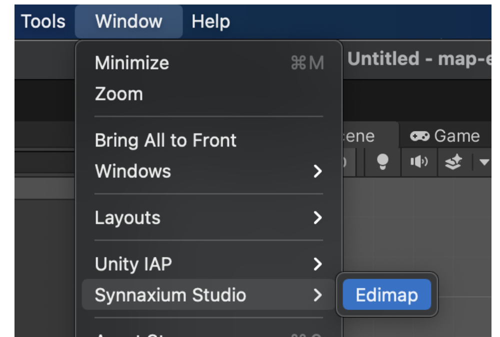
➔ the tile of tiling A will not have a border,
➔ the tile of tiling B will have a border (if tiling B has no reference to tiling A).

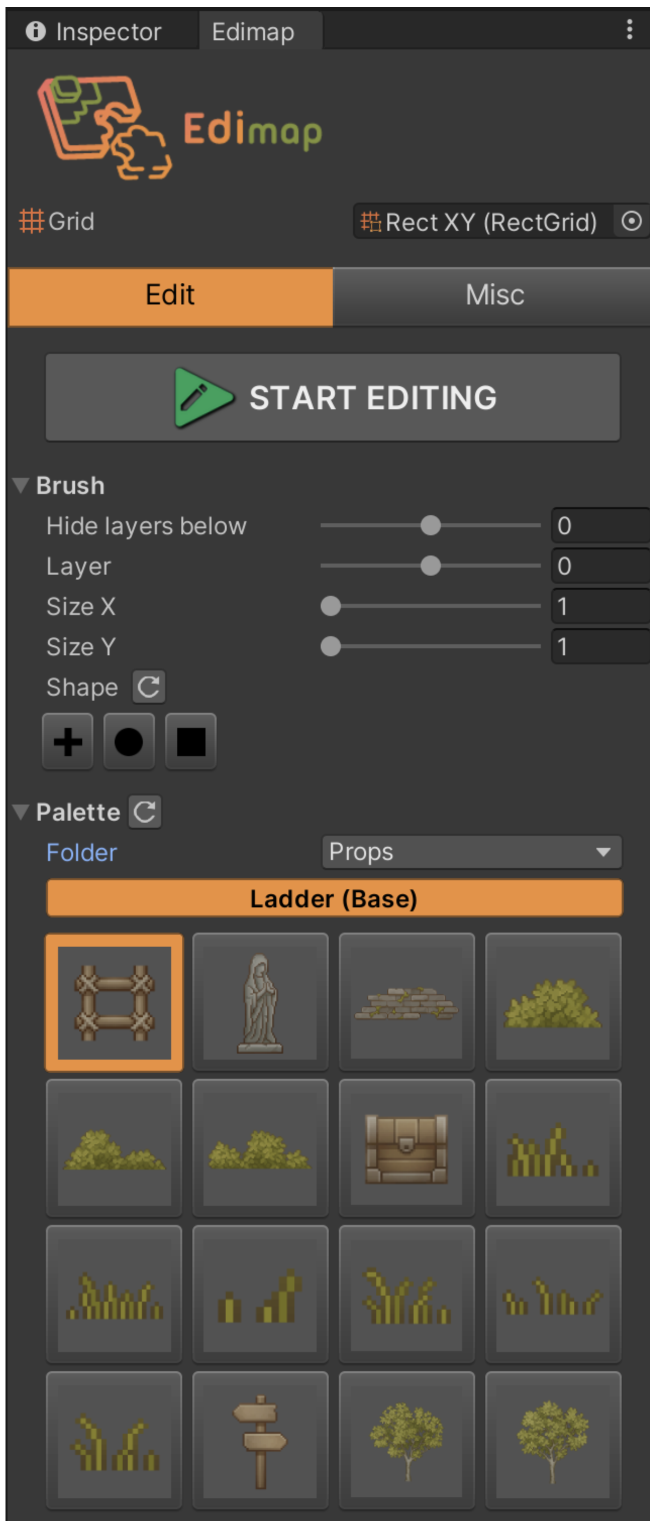
Synnaxium Studio

The Edimap window

You can open a new Edimap window from the **Window** menu:

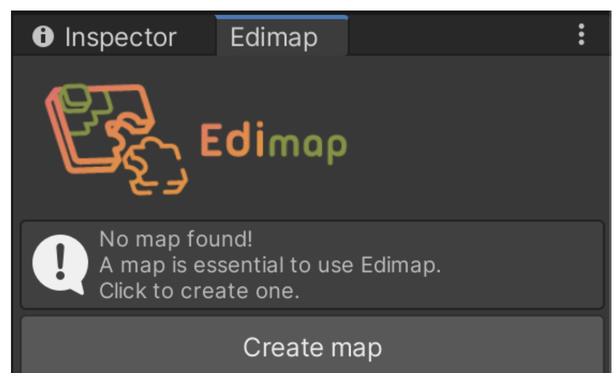
Window ➜ Synnaxium Studio ➜ Edimap





ℹ Only one window can be displayed at a time.

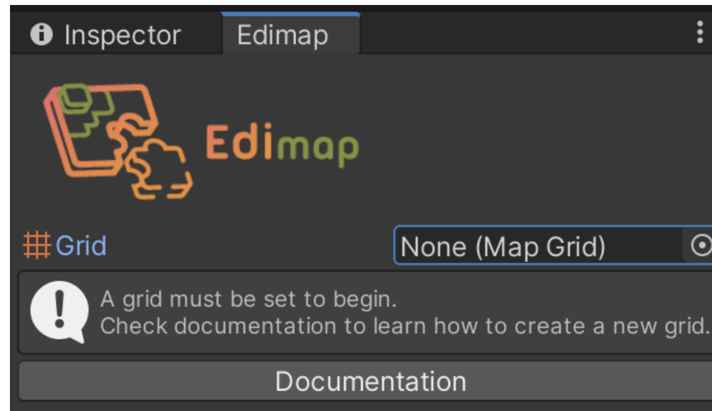
If no map has been created in the scene, a **Create map** button will appear in the window.



The latter allows you to create a new *GameObject* "Map" in the *SceneView* hierarchy. This object will be used as reference of the map by Edimap, it will include all the prefab and configurations added by the edit mode.

Grid

This field is essential to be able to use Edimap. If no reference has been filled in, then an information message tells you that this field must be filled in.



i A default grid is referenced in this field when creating a new map.

The grid must be set up when the map is first created and before editing it. This grid must not be modified afterwards. For more information on how to configure a grid, see the chapter MapGrid: Grid configuration.

Edit

This tab allows you to access the editing parameters as well as the palettes defined by the grid, but it also allows you to activate or deactivate the editing mode.

Brush

These options define where and how to edit the map, they allow you to manage the brush parameters.

Hide Layer Below

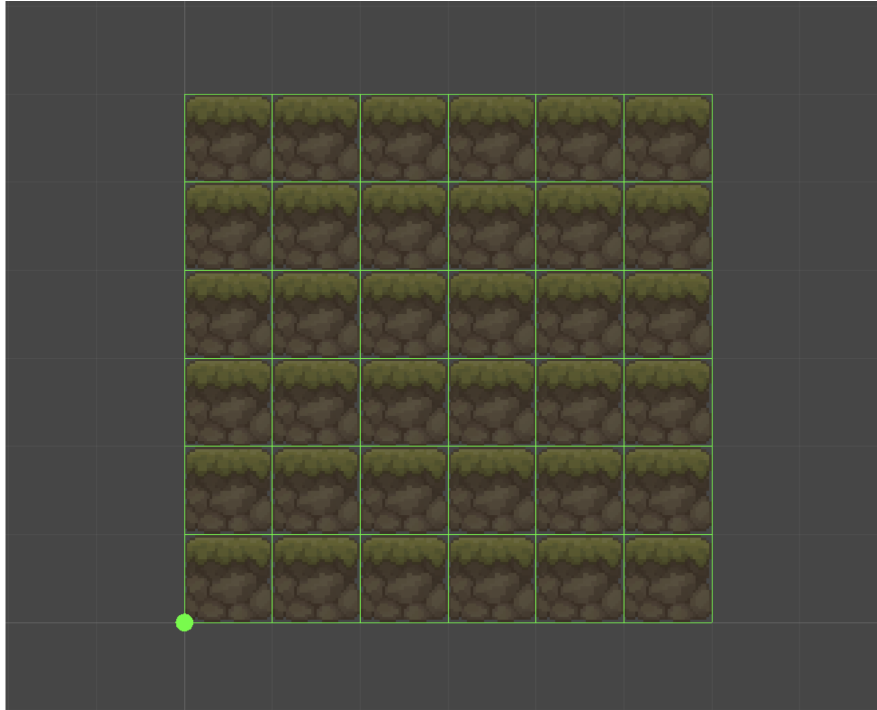
Layers lower than the value defined by this field will be invisible. This avoids some uncomfortable situations such as when you want to add an object on the first floor while the second floor blocks the view.

Layer

Defines the desired depth of the object to be placed on the map.

Size X & Size Y

This is the size of the brush used to place or remove multiple objects at once.



ℹ When a brush of several cells is used, the tiling is not resolved within the preview.

Shape

Select the brush shape you are using.

ℹ You can implement new shapes by inheriting the **BrushShape** class. All basic shapes are available in the folder:
Assets/Synnaxium/Edimap/Scripts/Impl/BrushShapes

⚠ **Once you added or deleted it is recommended to click on the ⟳ button to refresh available shapes.**

Palette

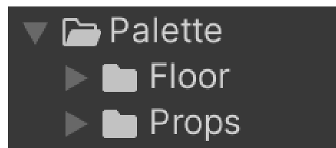
This part gathers all the palettes and their elements that can be placed on the map.

i A button to refresh the palette is available, it allows to take into account the modifications of a palette or of a prefab in case it has been modified.

Folder

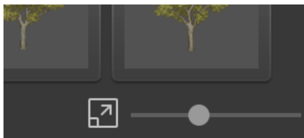
This is a list of the different palettes available. Simply select one of them to access its contents.

As a reminder, these are the sub-folders directly under the main folder of the palette.



Object grid

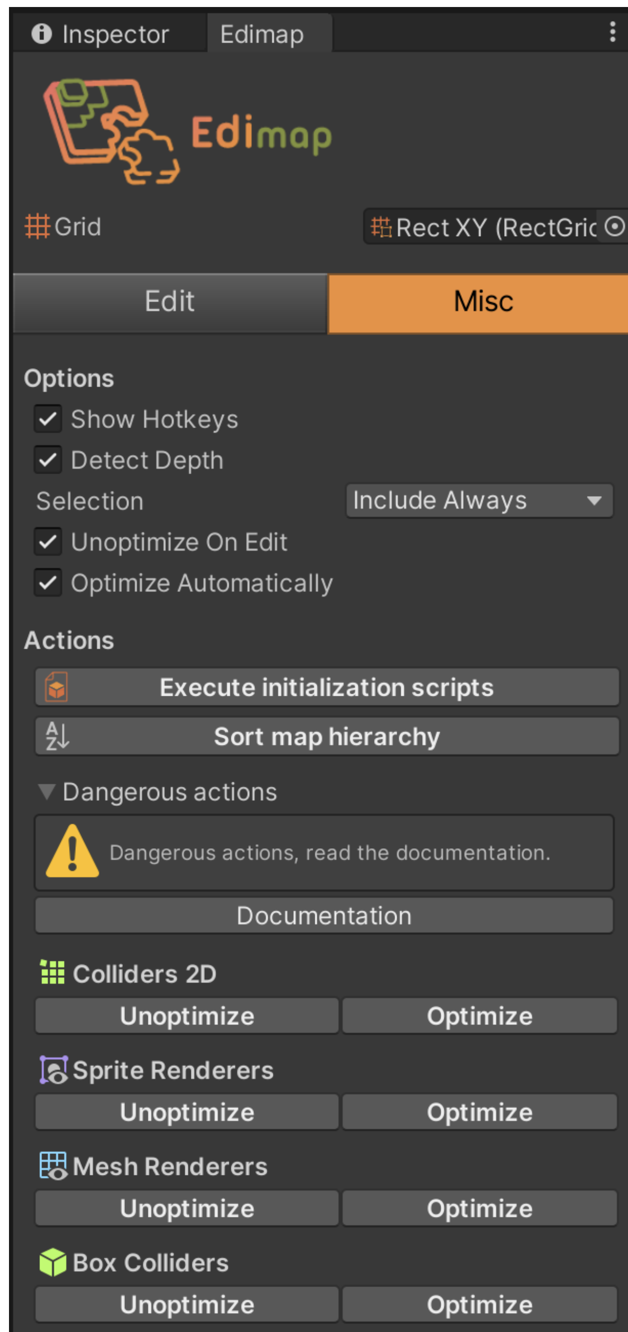
The list of available prefabs from the current palette that can be selected to be placed on the map.

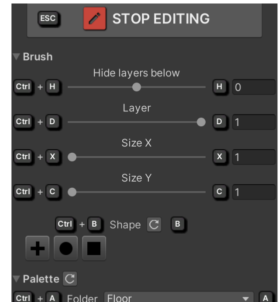


i At the bottom right, a slider allows you to vary the size of the preview boxes of the prefabs in the grid.

Misc

Provides access to additional options and tools.





Show Hotkeys

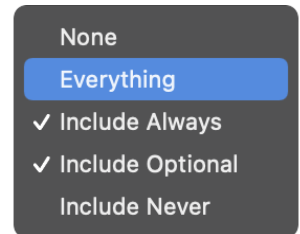
If this option is enabled, the shortcut icons are visible on the interface when editing mode is enabled, to remind the user of them.

i The shortcuts are customizable, if you want to know more see the Hotkeys chapter.

Detect Depth

Enables automatic detection of the layer on which to add the selected prefab when the edit mode is activated. However, as soon as the **Layer** value is changed, this option is temporarily disabled until a new edition is started.

i This feature is used when several layers are available in the used grid, i.e. mainly when editing 3D maps.



Selection

Allows you to choose what type of objects are allowed to be selected in the *SceneView*. It is possible to choose more than one type, or none by selecting *None*.

Unoptimize On Edit

Refers to the Unoptimize On Edit parameter of the grid currently referenced in Edimap.

Optimize Automatically

Refers to the Optimize Automatically parameter of the grid currently referenced in Edimap.

Execute initialization scripts

This button executes all initialization scripts present in the *SceneView*. To learn more about initialization scripts, refer to the chapter Execute a script while editing.

Sort map hierarchy

Rearranges the entire *GameObject* "Map" hierarchy in alphabetical order. This allows to find more easily an object present in this *GameObject*.

Dangerous actions

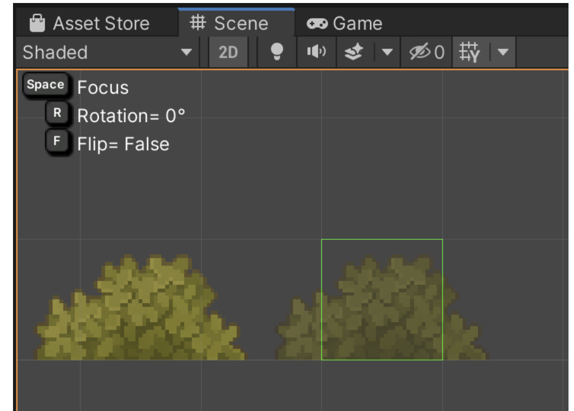
These actions can potentially affect your map. Currently, this section only contains optimization actions and their inverse, so we refer you to the Optimization section.

SceneView

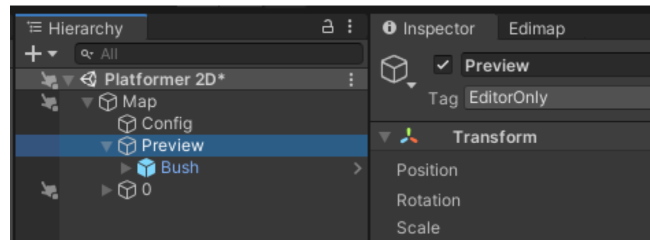
Preview

When the edit mode is activated, several changes can be noticed:

- ➔ an orange outline around the *SceneView*,
- ➔ indications on the placement of the prefab,
- ➔ an outline around the prefab of the defined size,
- ➔ a transparent version of the prefab to visualize the future rendering.

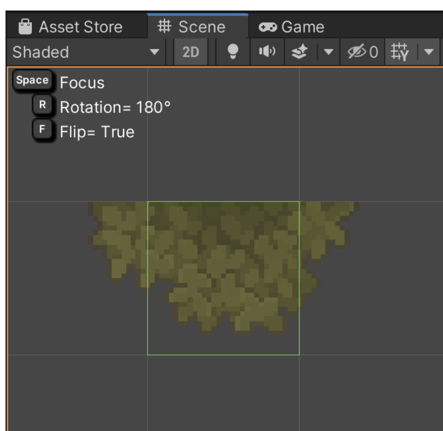


ℹ The visual of the prefab is made by creating a copy of it but without any *MonoBehaviour* component. So the possible *Awake* methods, called in editor script via the *ExecuteInEditorMode* attribute will not be called



Focus, Rotations & Flip

When editing the map, it is possible to focus on the preview, rotate or invert the prefab.



You can see in this example an illustration if a 180° rotation and inversion, keyboard shortcuts are used to perform these actions.

Hotkeys

Edimap provides a set of shortcuts for various actions in order to speed up the editing of the map.

⚠ **These shortcuts are only functional when the edit mode is activated.**

The list of default shortcuts is as follows:

Raccourci	Action	Mémotechnique
Escape	Exits edition mode	
Space	Focus on the preview	
D	Increases the layer	D for Depth
Ctrl+D	Decreases the layer	D for Depth
H	Increases the hidden layer	H for Hide
Ctrl+H	Decreases the hidden layer	H for Hide
X	Increases the brush size on X	X for X axis
Ctrl+X	Decreases the brush size on X	X for X axis
C	Increases the brush size on Y	Is next to X
Ctrl+C	Decreases the brush size on Y	Is next to X
B	Next brush shape	B for Brush
Ctrl+B	Previous brush shape	B for Brush
V	Next prefab	
Ctrl+V	Previous prefab	
A	Next palette folder	
Ctrl+A	Previous palette folder	
F	Flips the prefab	F for Flip
R	Rotates the prefab	R for Rotation

ℹ You can customize these shortcuts by creating your own global configuration of Edimap (see Global configuration). The **Hotkeys** group of the latter contains all the available shortcuts and a button to restore the default values.

Execute a script while editing

When a new prefab is placed on a map via Edimap, it is possible to run a script to manage an automatic configuration, such as:

- Ordering the sprites in a top-down 2D game.
- Automatically connecting a pair of teleporters.
- Automatically connecting a lever to a door.
- etc.

Since Edimap takes care of instantiating the prefabs, we have at the same time the opportunity to execute code without depending on the *ExecuteInEditMode* attribute.

MapObjectInit : Folder configuration

Within your **PaletteSettings**, you will find an **Inits** array that can contain *ScriptableObject* of type **MapObjectInit**.

You can create your own implementation by inheriting this script. For example, the code below changes the color of the tiles according to their depth:

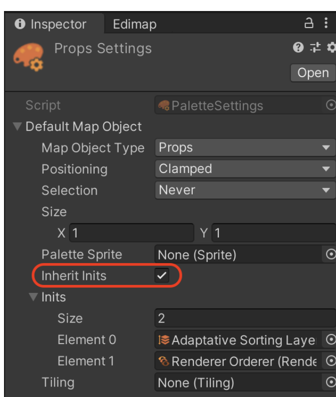
```
using UnityEngine;

namespace Synnaxium.Edimap
{
    [CreateAssetMenu(menuName = "Synnaxium Studio/Edimap/Map Object Inits/Depth Color")]
    public class DepthColor : MapObjectInit
    {
        [SerializeField] private float distance = 5f;

        public override void InitGameObject(MapGrid grid, MapObjectSettings settings, GameObject gameObject)
        {
            var renderers = gameObject.GetComponentsInChildren<SpriteRenderer>();

            foreach (var renderer in renderers)
            {
                renderer.color = Color.Lerp(renderer.color, Color.black, renderer.transform.position.z / distance);
#if UNITY_EDITOR
                UnityEditor.PrefabUtility.RecordPrefabInstancePropertyModifications(renderer);
#endif
            }
        }
    }
}
```

The *grid* argument is a description of the grid currently in use, while *settings* contains the configuration of the object (positioning, number of cells used, type of object, etc.).



ℹ The scripts will be executed in the order of presence within the array.

ℹ You will notice an **Inherit Inits** option within your **PaletteSettings**. If this option is checked, the folder inherits the initializations configured in the **PaletteSettings** of the parent folders.

IMapObjectInit : Object configuration

You can inherit the **IMapObjectInit** interface on one of your *MonoBehaviour*:

```
public class MyCustomCode : MonoBehaviour, IMapObjectInit
{
    public void MapObjectInit(MapGrid grid, MapObjectSettings settings)
    {

    }
}
```

Attach this *MonoBehaviour* to your prefab in a child's *GameObject*, adding the *EditorOnly* tag, and you're done.

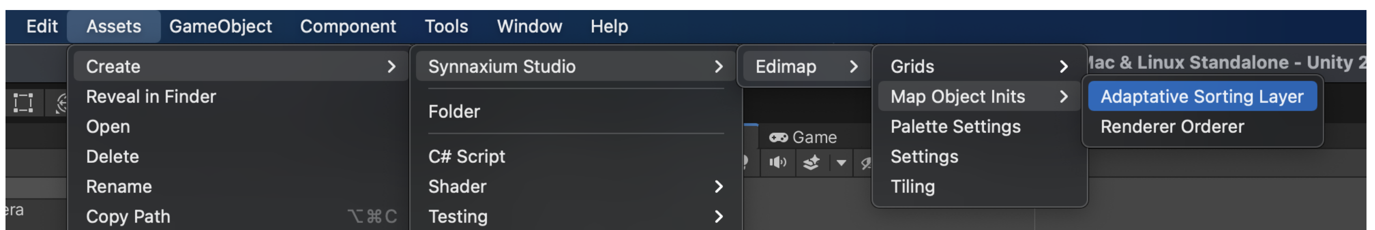
i This operating mode is more intrusive because a modification of the prefab is required, but the option is available if you have specific needs for a particular prefab.

AdaptativeSortingLayer

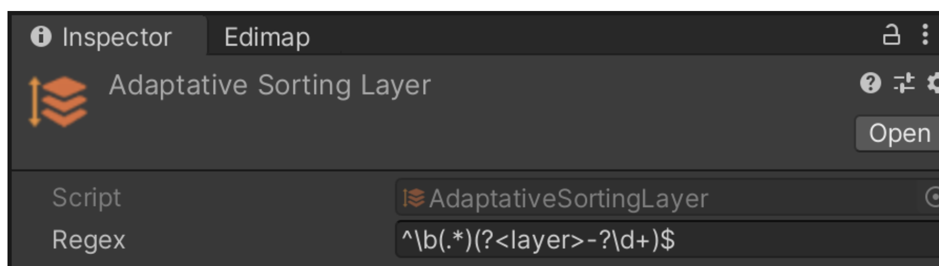
Edimap proposes an initialization process for objects dedicated to 2D maps with several depths, called **AdaptativeSortingLayer**.

This initialization script is available in the asset creation context menu:

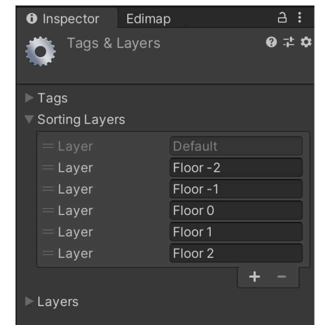
Create ➜ Synnaxium Studio ➜ Edimap ➜ Map Object Inits ➜ Adaptative Sorting Layer



The asset has a regular expression (regex) as a parameter that allows you to define the syntax of your *Sorting Layers*.

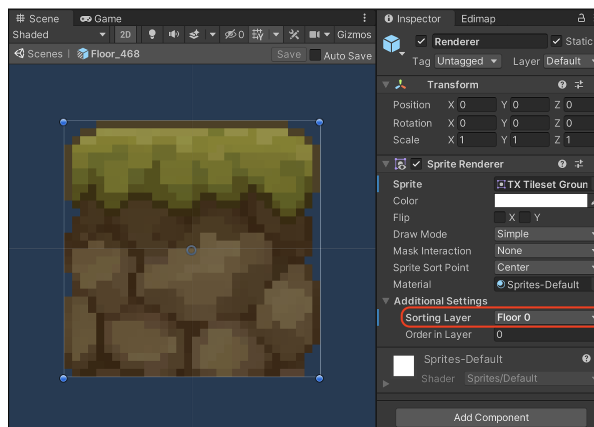


By default, this regex will interpret *Sorting Layers* that end with a depth number.



ℹ The regex must always retrieve the depth index under the group name "layer".

Reference this initialization in the **Inits** array of your **PaletteSettings** and make sure that your prefabs use one of the numbered *Sorting Layers*.



Thus, **AdaptativeSortingLayer** will take care of matching the *Sorting Layers* according to the layer in which the prefab will be placed.

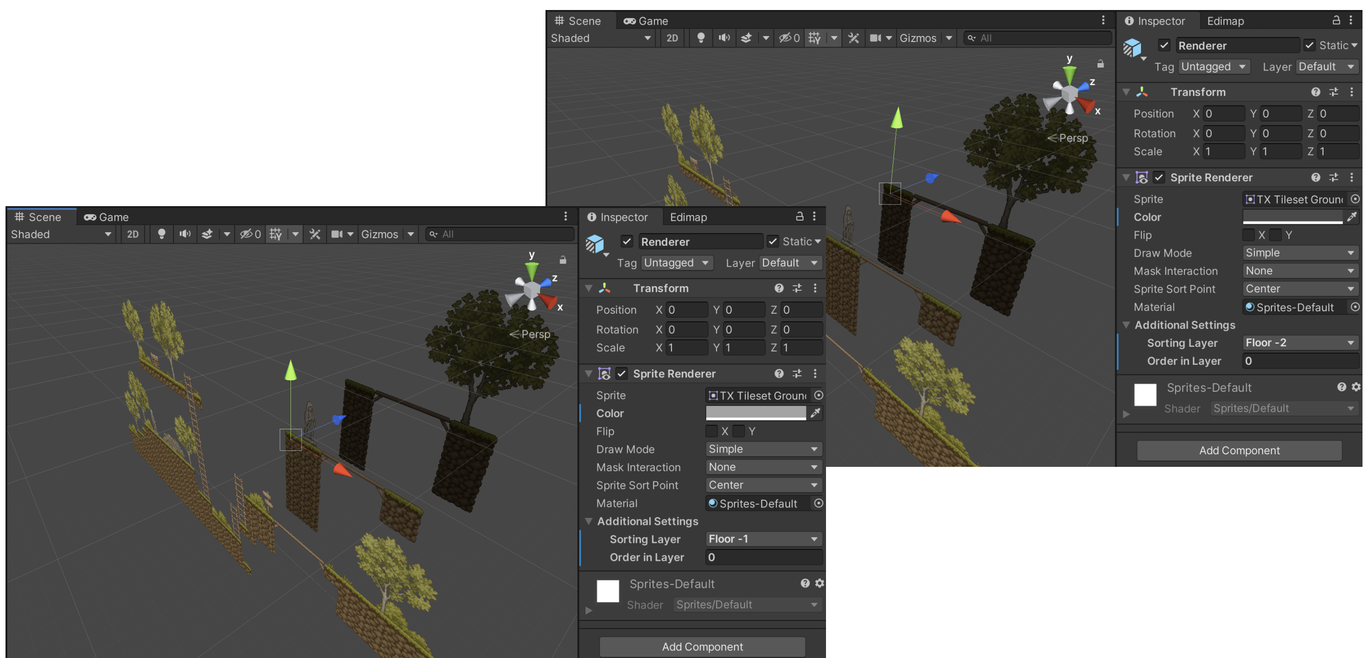


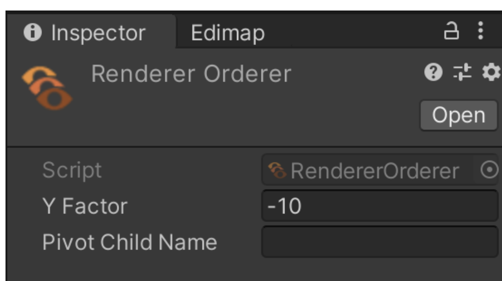
Illustration of the use of the example scripts DepthColor and AdaptativeSortingLayer.

RendererOrder

This initialization script is useful for 2D maps without using depth, especially for 2D top view or isometric games. It automatically defines the **Order in Layer** parameter of the tiles renderers according to their position.

Like the previous script, this one is available in the asset creation context menu:

Create ➜ Synnaxium Studio ➜ Edimap ➜ Map Object Inits ➜ RendererOrder

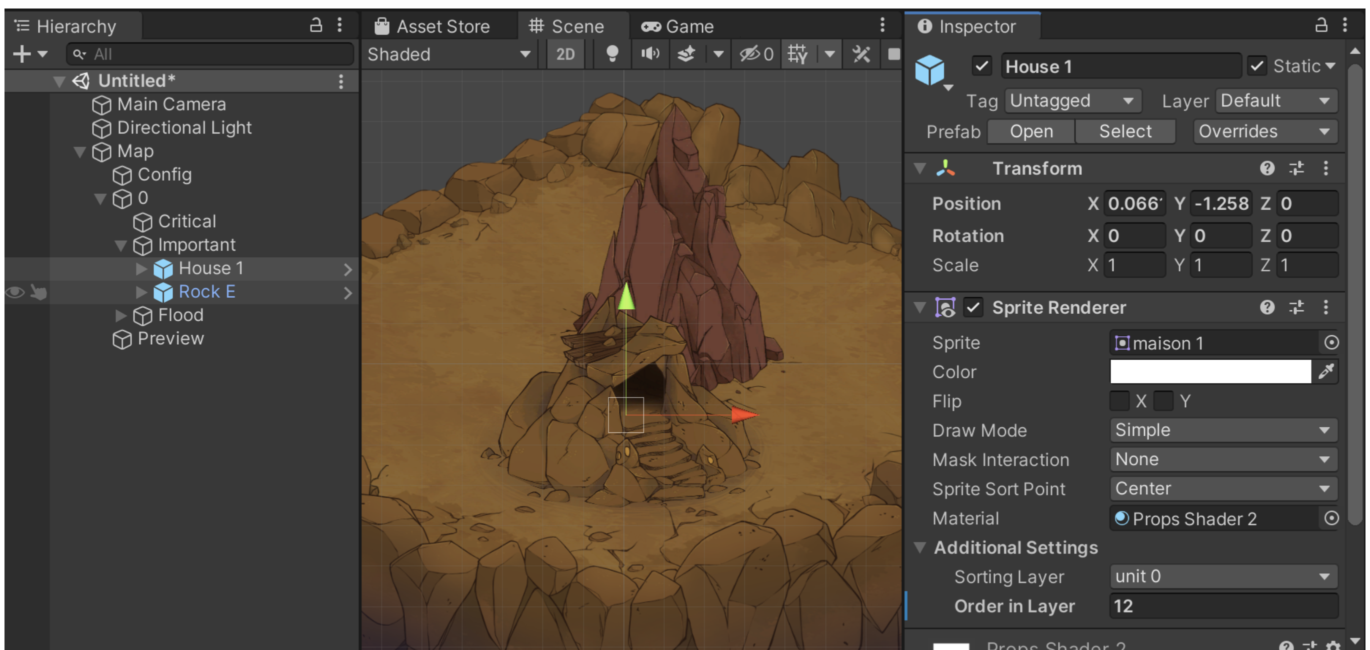


Y Factor

This is the factor by which the position of the sprite will be multiplied to obtain the sort order.

Pivot Child Name

If a child exists and the name of the child matches, then the script will use the *Transform* position of the child instead.



In the example above, *House 1* has a higher **Order In Layer** than *Rock E*, so it is displayed above the latter. Yet both objects have the same **Sorting Layer**.

Optimization

Edimap uses a so-called prefab workflow, which means that a prefab is instantiated for each cell of your game.

The huge advantage of the approach is that it allows you to have concrete concepts for each part of your map: lava, doors, creatures, checkpoints, etc.

A classic tilemap would only give you an abstract representation of the environment, and you would still have a lot of work to do to get a playable map. This is not the case with Edimap, editing a map gives you directly the finished product.

Obviously, it is not possible to deliver a game where each cell is a prefab, there would be tens of thousands of renderers and colliders, it would drop your performance.

The solution is to work in prefab, but to optimize the final rendering by merging the renderers and colliders together in a second step.

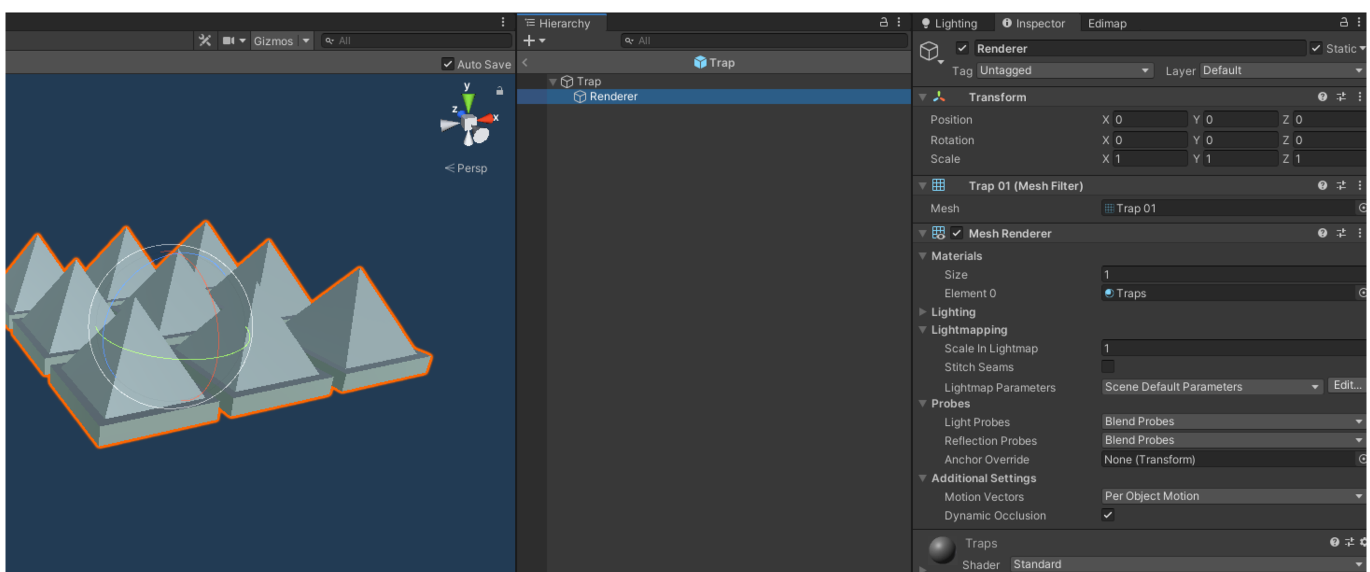
Prefab design

When you design prefabs, several classic use cases can appear:

- The prefab is composed of a single renderer or a single collider.
- The prefab mixes several notions between rendering, collision and gameplay.

In the simple case, there is no special consideration other than to ensure that the prefab is set as **Static**.

For prefabs mixing several responsibilities, it will be necessary to decompose into children *GameObject* to allow Edimap to isolate the components.



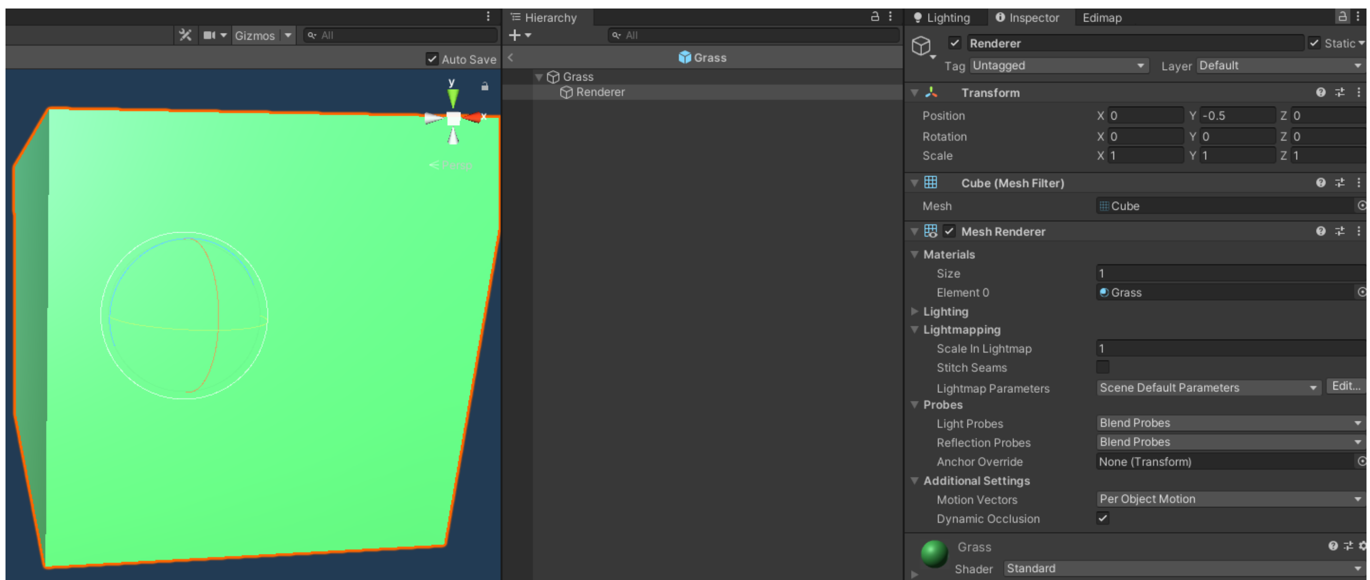
For example, with the traps in the previous example, we will have two features. They have a rendering with a *MeshRenderer* and will hurt the player if they step on them.

When Edimap will optimize this prefab, the tag of the *GameObject* containing the *MeshRenderer* will become *EditorOnly*, so this object will not be present in the build.

However, we do not wish to lose the part that inflicts damage to the player.

To manage this, we need two *GameObject*:

- a parent who will take care of the damage,
- a child who will take care of the rendering and who will be potentially eliminated from the build by Edimap.

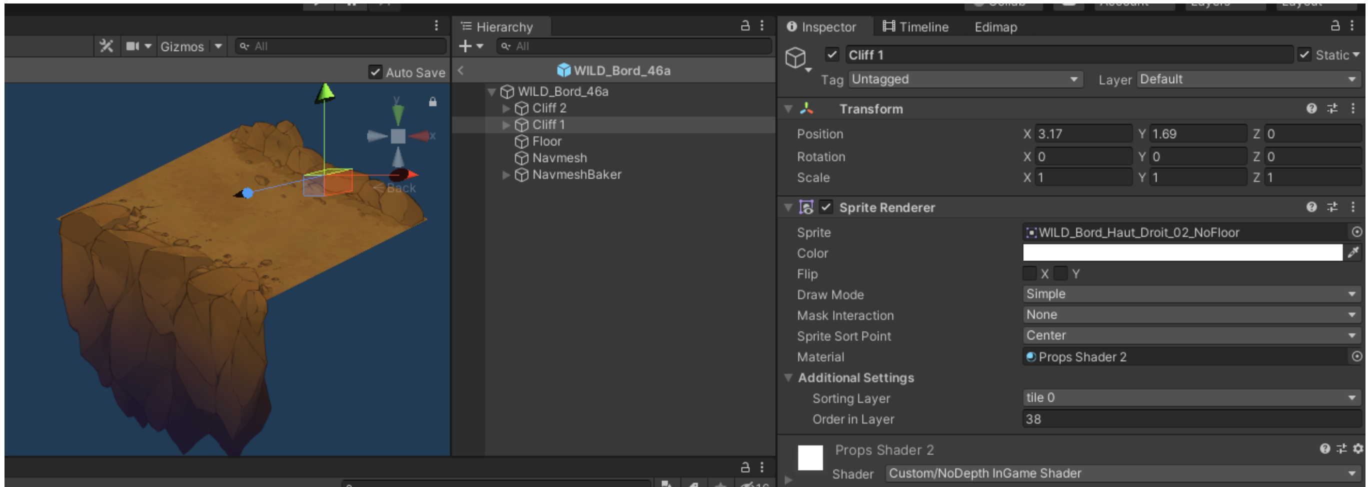


An inherent advantage of this approach is the management of your pivots. The renderer position can be adjusted easily by simply moving its local position.

Here this object "*Grass*" will be eliminated from the build entirely, because the root is the collider which will be optimized.

Even without the excuse of the *Collider* on the parent object, it would have been quite possible to simply put the root in *EditorOnly* anyway, knowing that we intend to optimize the *MeshRenderer* in any case.

CAs our object will be converted into another mesh by Edimap, we have the advantage of being able to add superfluous *GameObject* simply to correct the positioning of our objects, which allows us to limit the round trips between the integration teams and the artists.



In the example above, our floors could quickly become hell to integrate. Here the ground is composed of three parts that use two materials, which forces us to separate them.

Isometric rendering also imposes some particular constraints on the sprite positions in order to automatically generate a correct rendering order.

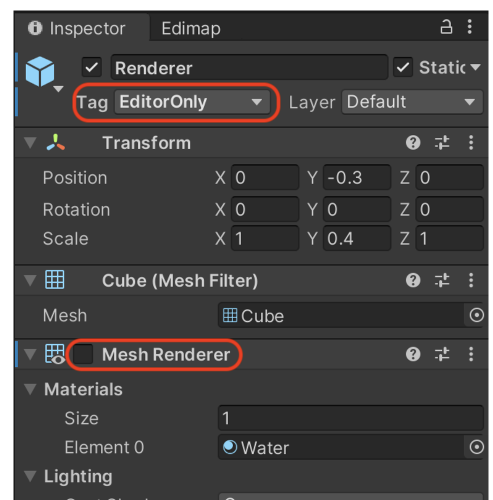
A possible solution would be to impose on designers export in which the pivot is automatically placed in the right place for each part. However, this creates a lot of unnecessary transparency in the image files, and often wastes substantial time for designers who do not necessarily have a perfect understanding of the technical constraints.

Here we can simply place the parts in the prefab ourselves with no extra costs involved.

Technical details

When you optimize a part of your map, be it *SpriteRenderer* or *BoxCollider*, Edimap does the optimization in a similar way.

First, the relevant objects are switched to *EditorOnly* and the optimized components are deactivated.

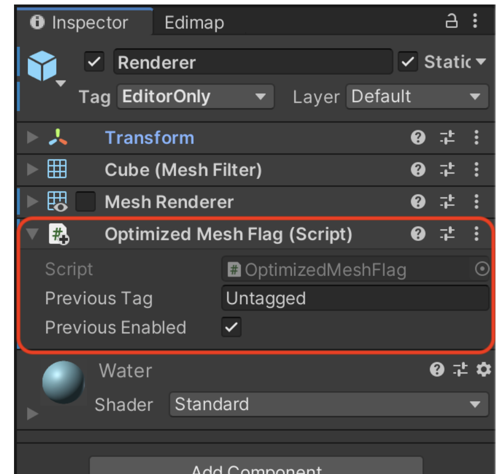


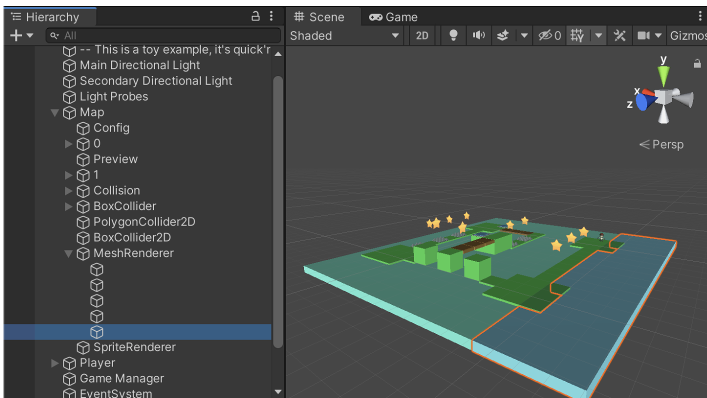
Secondly, a component is also attached to the object so that Edimap can remember that the object has been optimized.

In case you remove the optimization, this allows Edimap to restore the old values.

⚠ **Never delete this component yourself.**

ℹ We have chosen to leave it visible so that you can quickly identify why one of your components is disabled.



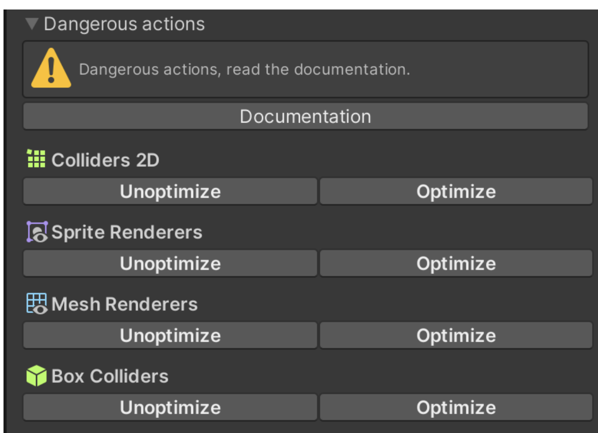


Finally, Edimap will add new objects within your map, in a section with the same name as the optimized component.

You will then see a set of *GameObject* with no name, where each object is a grouping optimized renderers or colliders.

Usage

From the Edimap window, it is possible to access the **Dangerous actions** section with the **Misc** tab. This one groups the different possible optimizations with two buttons, one to optimize the map and another one to remove the optimizations.



ℹ Automatic optimization or un-optimization parameters are available via the MapGrid *Scriptable Object* or in the Misc part of the Edimap window.

ℹ To automate the process on a set of scenes, you can use the Edimap API via the **BoxColliderOptimizer**, **SpriteRendererOptimizer**, etc. classes.

SpriteRenderer

SpriteRenderer optimization follows the same rules as Unity dynamic batching.

The conditions for merging two *SpriteRenderer* are:

- Same material with the same parameters.
- Rendering order that is not separated by a third incompatible *SpriteRenderer*.

Colliders 2D

For the *BoxCollider2D*, optimization is currently greedy, which means that the solution is not strictly optimal, but largely sufficient.

Two *BoxCollider2D* will probably be merged if they can be perfectly encompassed by a single *BoxCollider2D* by adjusting the center and size parameters.

For the *PolygonCollider2D*, we use the *Clipper* implementation which guarantees an optimal solution. However, *PolygonCollider2D* inherently have poor performance within Unity. It is therefore to be used in moderation, it is preferable to use *BoxCollider2D* as much as possible.

MeshRenderer

All *MeshRenderer* will be optimized together if they use the same material with the same parameters.

BoxCollider

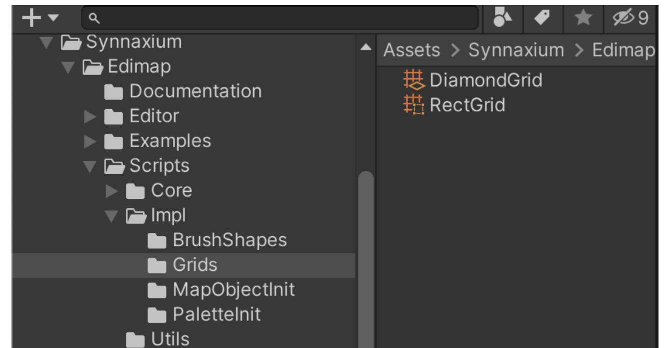
Like the *BoxCollider2D*, two *BoxCollider* will be merged if it is possible to encompass them exactly with a larger *BoxCollider*. The geometry is perfectly preserved, but the solution is not necessarily optimal. However, the quality of the algorithm is largely sufficient.

Customized grid

How it works

Edimap offers you two default grid types: the rectangular grid and the diamond grid.

The rectangular grid allows you to make squares and any rectangular shape, while the diamond one allows you to make isometric cells (an isometric grid is composed of diamonds with cell sizes of 2 by 1).



These two grids may not correspond to your expectations, so Edimap provides an API to create your own implementation. To do so, you just have to create a *ScriptableObject* script inheriting from **MapGrid**, and fill in the abstract methods. Once your grid is created, you just have to reference it in the Edimap window to use it.

i Feel free to copy the code of Edimap's two basic grids for inspiration.

Coordinates

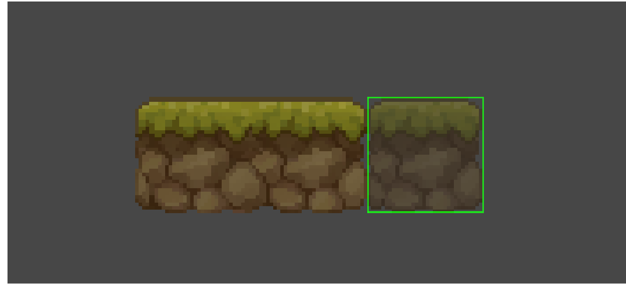
The Edimap grid always works with discrete coordinates which are represented by *Vector2Int*.

Each cell within your grid must therefore be identified only by a pair of integers x and y. Your grid implementation will mainly consist of establishing a function to move from this pair of identifiers to a coordinate in space and vice versa.

i Note that the notion of XY and XZ grid is not present in this API. To simplify your life, this transposition is made separately.

OnWorldToCell

When you point with your cursor within the scene, Edimap infers a selected cell. This is what allows you to place objects side by side without overlapping.



Parameter	Type	Description
position	*Vector2*	The spatial position to be translated into coordinates.
return	*Vector2Int*	The coordinate to identify this cell.

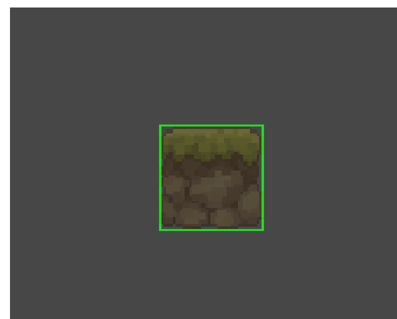
OnCellToWorld

When Edimap wants to place an object in the scene, it will place this object on the selected cell. It is then necessary to translate this coordinate into a point in space within the scene.

Parameter	Type	Description
cell	*Vector2Int*	The coordinate of the cell.
return	*Vector2*	The center of the cell.

OnDraw

Called when Edimap wants to display a preview of the grid for the user. The purpose of this method is to return a list of points used to draw the lines of a cell.



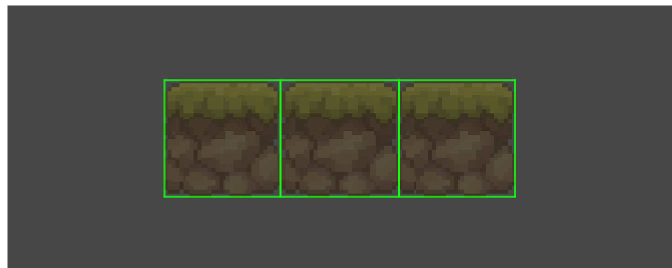
OnDraw *describes the green lines of the preview*

The points are to be provided two by two within the list. Each pair of dots forms a line in the scene.

Parameter	Type	Description
objectSize	*Vector2Int*	The size of the cell to be displayed (for objects that extend over several cells)
cell	*Vector2Int*	The cell to preview. If the size of the object is more than one cell, it is the bottom left most cell (smallest x and y indices).
z	*float*	The depth at which the preview takes place. This value must be injected into the z-component of the points in the list.
return	*Vector3[]*	A table of points. Each pair of points will give a line.

OnCellDelta

OnCellDelta is used to define the distance between two cells when incrementing the coordinates.



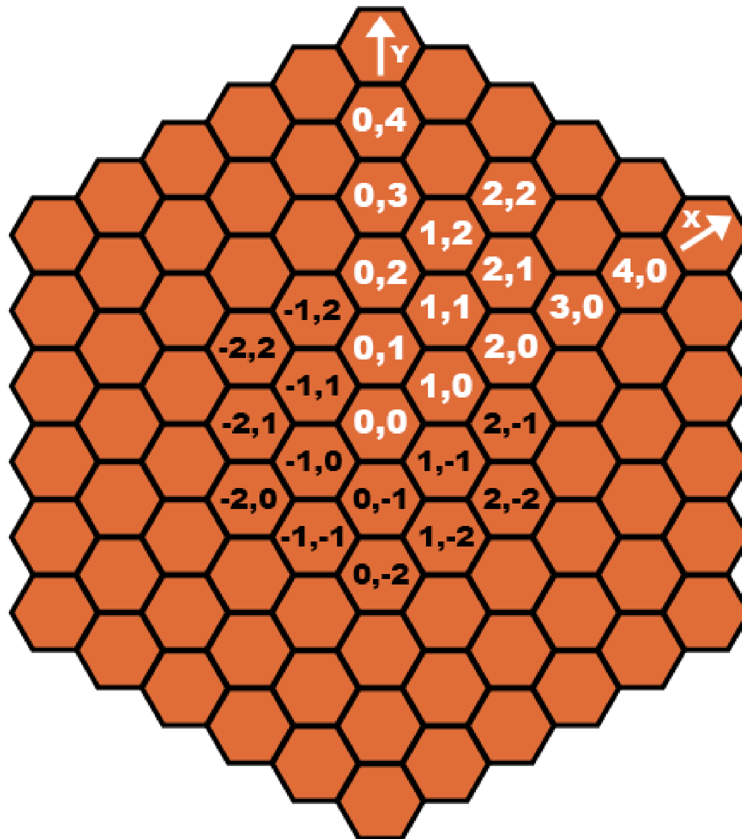
*The brush uses **OnCellDelta** to offset the previews.*

Parameter	Type	Description
x	*int*	The X coordinate shift
y	*int*	The Y coordinate shift
return	*Vector2*	The resulting shift in spatial coordinates

Example: Hexagonal grid

Currently, the hexagonal grid is not yet implemented. Its integration is planned in the near future depending on the interest of Edimap by the Unity community.

In the meantime, we can still quickly explain the process by schematizing the grid with its axes and some coordinates:



The hexagonal grid can sometimes be scary, with its 6 neighbors per cell.

Mathematically, from Edimap's point of view, it is a very common 2D grid. The only real subtlety is the positioning of the X axis which is 30° from the usual direction.