PROCEDURAL GENERATION GRID
USER MANUAL

About Procedural Generation Grid

- Procedural Generation Grid is a system which uses grids to define fields and paths for procedural generation algorithms.

- System is designed to be flexible and able to generate very various and complex objects/places. **Have full control on the procedural generation logic!**

- It can be used to generate procedurally whole game levels at runtime or for editor use to quickly paint levels for speed up level design process.

- System is using something like visual scripting, so you don't need to know how to code (but you can still take benefits out of it) you just need to plan logics for your needs.

- Procedural Generation Grid is right now **in Beta Stage** so, **many features may change**, **new features will come** and some small elements are not fully finished but the developmental stage is completed enough to be functional in many cases.

- Package is providing additional packages for randomly placing smaller objects in physical space using Unity's collision system (Object Stamper and Pipe Generator)

- Package is not dedicated to generating giant worlds (it still can work like that), it's focusing on smaller detailed areas like room interior with furniture, tunnels with many elements, presets for painting modular walls etc.

Contact and other links you will find in Readme.txt file

Index

1: The Procedural Grid Generation System

Grid Generation Process:

Before working with Procedural Generation Grid you need to know what are the fundamentals of it's logic.

System is <u>running spawning rules for every cell of the grid</u> provided by the user. <u>Order of provided grid cells modificators are very important</u>!

For example: when you first spawn walls and you want to create doors you must tell doors modificator to remove walls out of its place, but when you first create doors then you must tell wall modificator to ignore cells with doors spawned.
(it's just an example, there are prepared tools to set up doors clearly)

You must be careful when setting up spawning logics to avoid logic conflicts, depending on what you want to create, sometimes logics can get more complicated, keep that in mind.

<u>Field Setup</u> is the main asset type of the whole package, it contains groups (packs) of grid modificators which are spawning or removing objects generated on grid.
You can group modificators, which you can find very useful, for example you can make a group of field modificators which spawns the whole interior room with props inside, or do separate Groups of Modificators, one with just walls, floors, ceiling etc. other with light lamps, or other cosmetics and another Modificators Group with props like furniture, debris etc.
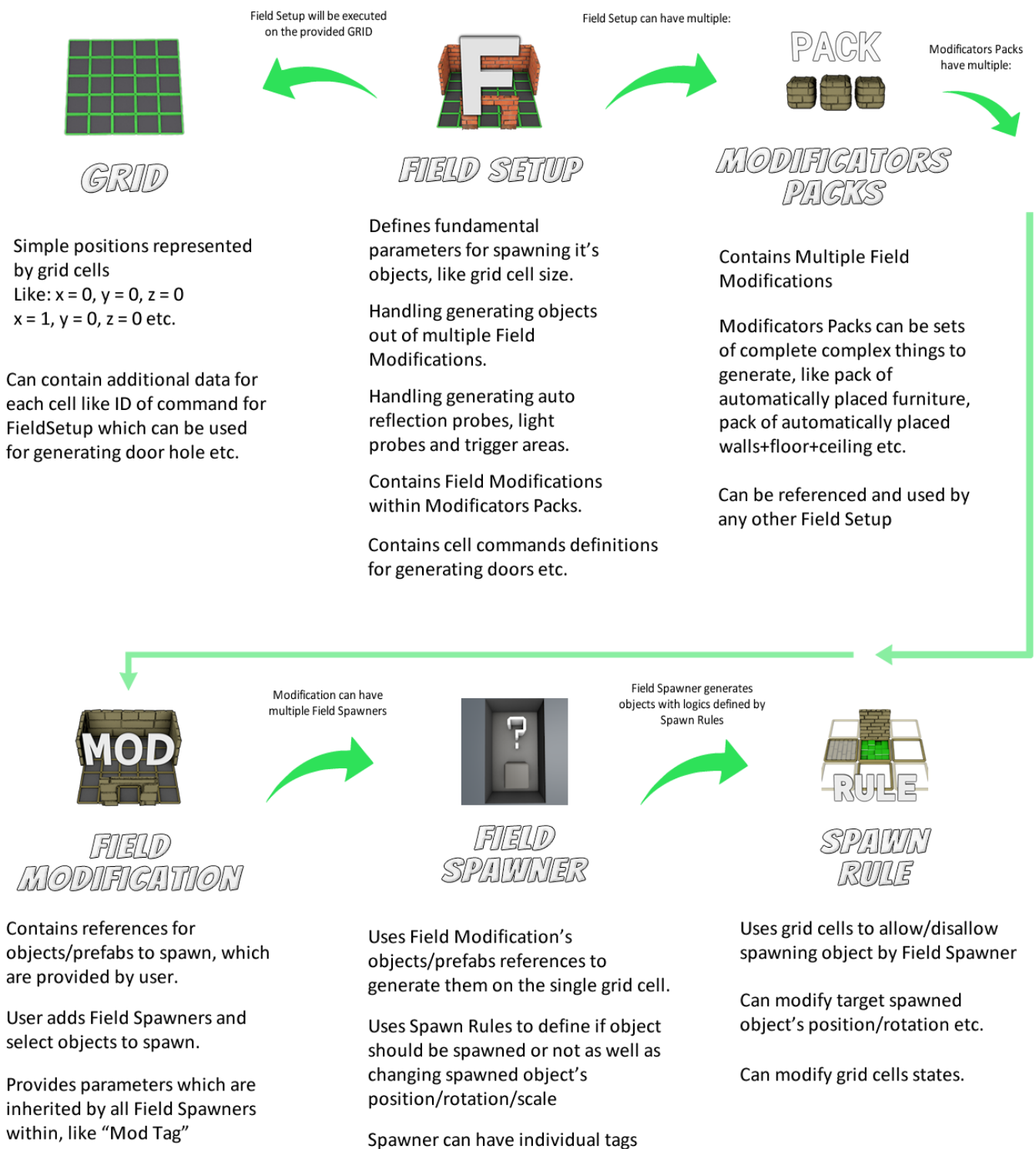Then in Field Setup you can freely assign groups to be executed, for example group of walls as first, then run cosmetics group and in the end select your props modificators group.

<u>Field Modificators</u> can have variant files, variants can be created using the "V" button which you can find in Field Designer Window or Inspector window of Modificators Pack. Variant means that Field Modificator is using all rules of parent Field Modificator but can change prefabs setted inside it, so you don't need to set up rules multiple times when you just need to change the look of some walls as an example.

Procedural Generation Grid system is not generating game objects during cell checking/spawn rule execution process, it spawns them at the end, all spawn data is contained within additional classes which are used after all Field Setup's rules are computed.
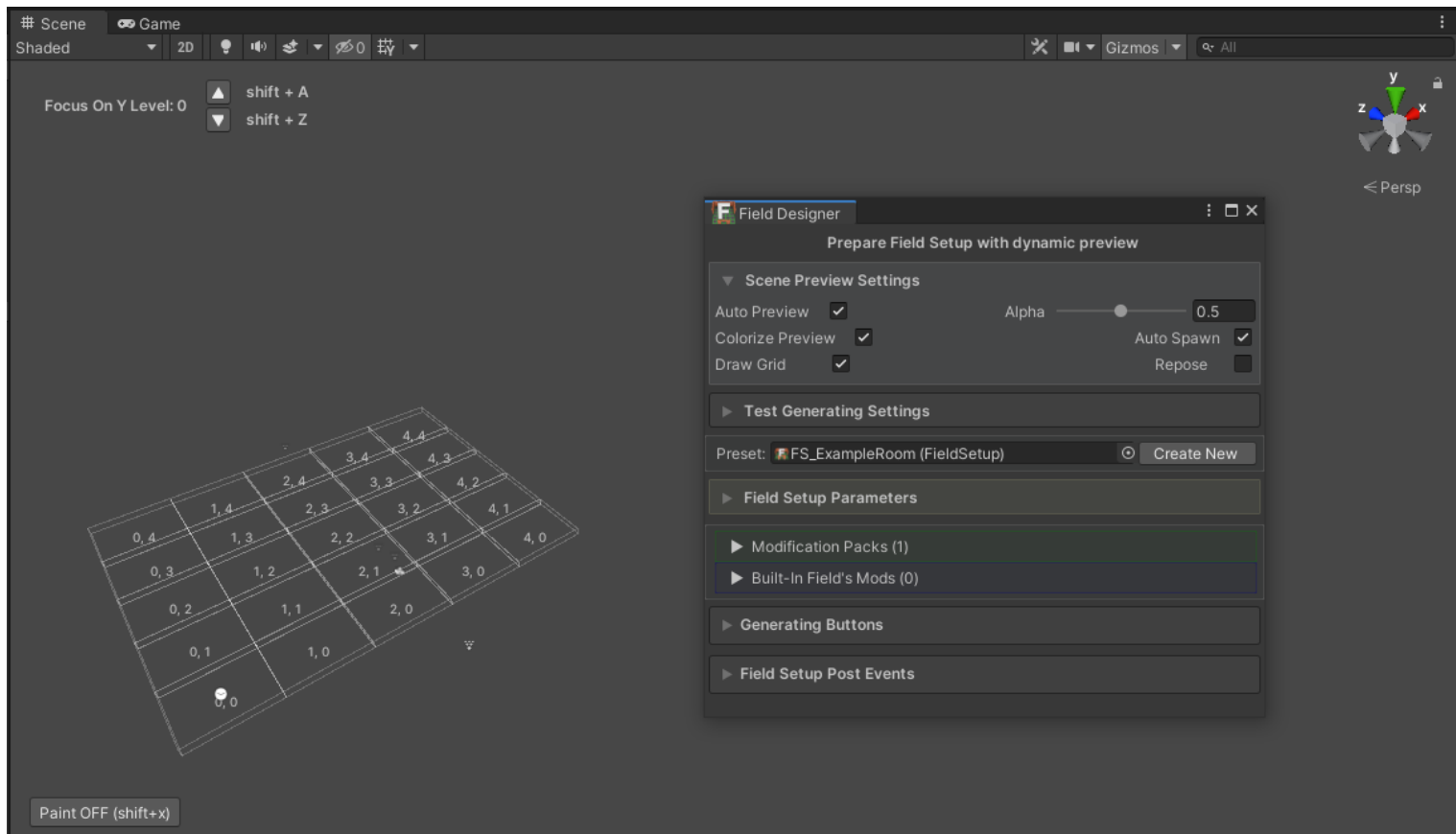
Field Setups/Presets:

Package offers preset structure which can help you a lot.
Take a quick look on relations between different instances:

Field Setup will be executed
on the provided GRID

Field Setup can have multiple:

Modificators Packs
have multiple:

GRID

Simple positions represented
by grid cells
Like: x = 0, y = 0, z = 0
x = 1, y = 0, z = 0 etc.

Can contain additional data for
each cell like ID of command for
FieldSetup which can be used
for generating door hole etc.

FIELD SETUP

Defines fundamental
parameters for spawning it's
objects, like grid cell size.

Handling generating objects
out of multiple Field
Modifications.

Handling generating auto
reflection probes, light
probes and trigger areas.

Contains Field Modifications
within Modificators Packs.

Contains cell commands definitions
for generating doors etc.

MODIFICATORS PACKS

Contains Multiple Field
Modifications

Modificators Packs can be sets
of complete complex things to
generate, like pack of
automatically placed furniture,
pack of automatically placed
walls+floor+ceiling etc.

Can be referenced and used by
any other Field Setup

Modification can have
multiple Field Spawners

Field Spawner generates
objects with logics defined by
Spawn Rules

FIELD MODIFICATION

Contains references for
objects/prefabs to spawn, which
are provided by user.

User adds Field Spawners and
select objects to spawn.

Provides parameters which are
inherited by all Field Spawners
within, like "Mod Tag"

FIELD SPAWNER

Uses Field Modification's
objects/prefabs references to
generate them on the single grid cell.

Uses Spawn Rules to define if object
should be spawned or not as well as
changing spawned object's
position/rotation/scale

Spawner can have individual tags

SPAWN RULE

Uses grid cells to allow/disallow
spawning object by Field Spawner

Can modify target spawned
object's position/rotation etc.

Can modify grid cells states.

In most cases you will use **Field Designer Window** which you can open through "Window->FImpossible Creations->Level Design->Grid Field Designer"

Field Setup is the root of what is spawned on the provided grid.

Field Designer Window is dedicated to focus on spawning logics, not on creating gameplay shaped grid. Debug grid visible when the Field Designer Window is opened will disappear and be forgotten after closing the window. **You should edit Field Setup files only through Field Designer Window (it opens automatically when double-click on Field Setup File)**

Scene Preview Settings:

Configure how you want preview grid generation on gizmos or with spawning prefabs on the scene (slower).

(**Repose** is changing position of Field Designer after scripts compilation)

Test Generating Settings:

Configure on what grids you want to test your spawning logics, you can simply generate random scaled rectangle grids, or paint on scene grid as you like or generate branched tunnels.
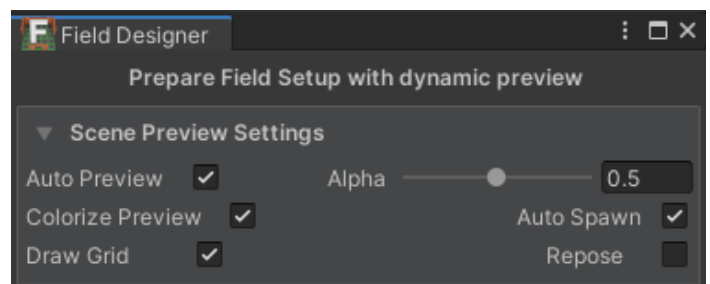
Grid will be generated each time you change any rule through the inspector window if **"Auto Spawn" is enabled** or you can trigger it with **"Generating Buttons"** lower in the Field Designer window.
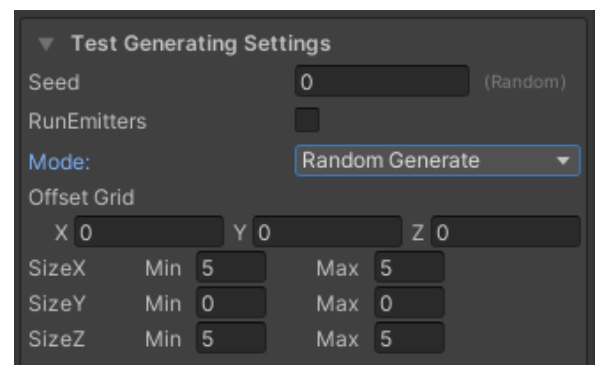
Field Setup Parameters:

Configure field to objects you will spawn on grid, **it's best to fit CellSize to floor/wall tiles** (but you can make it smaller and spawn floor/wall tiles with grid scaleUp features - advanced)
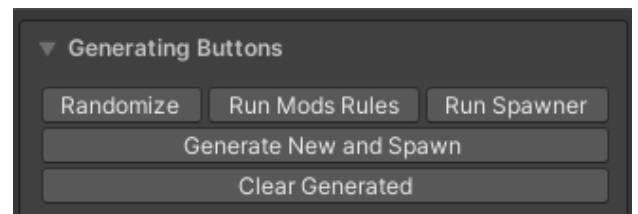
Cell Instructions Commands can be used when the grid on which Field Setup is used provides additional data on which cells commands should be used. It's really useful for making doors / door holes etc. but can also be used for many other things.
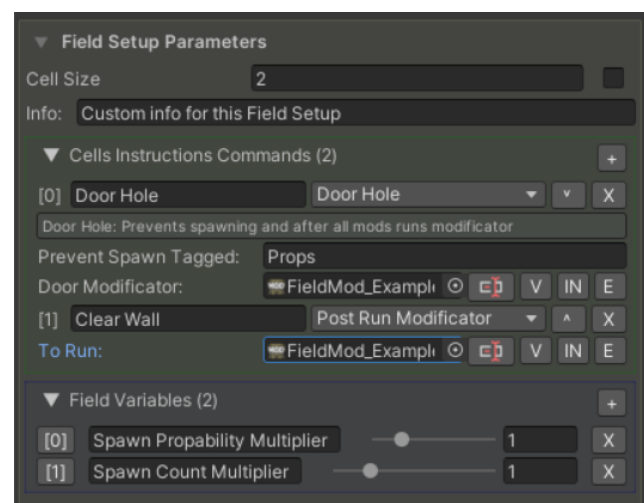
Field Variables can be used for controlling some rules parameters from FiledSetup level or through injecting (described later)
Some rules parameters have possibility to be used by Field Variable, you can identify them by a half white dot which must be pressed to assign variable:
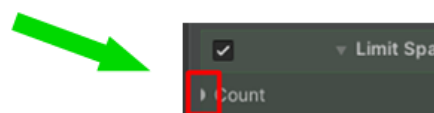
To work with a field designer you must create a FieldSetup file inside your project directory, the best place them in a new directory since you probably will create a few of them later.
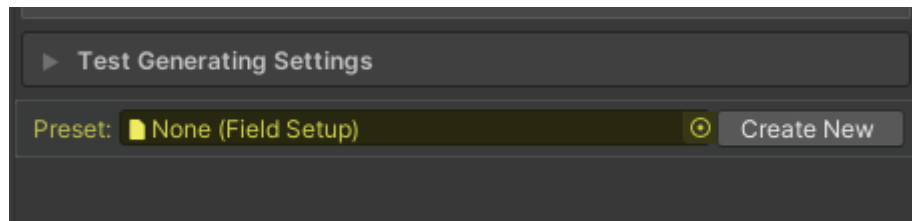

After creating the **FieldSetup** preset file you will see the **"Modification Packs" list**. (FieldSetup is created with a root (built-in) modificators pack but can use multiple other packs in a shared way)

Inside the box below there are displayed **Field Modificators** which are inside currently selected **Modificators Pack.**

After creating the preset there is just a root modification pack and no **Field Modificators**.

To know what are the buttons on the right, just enter on them inside the Unity Editor and a tooltip will appear.
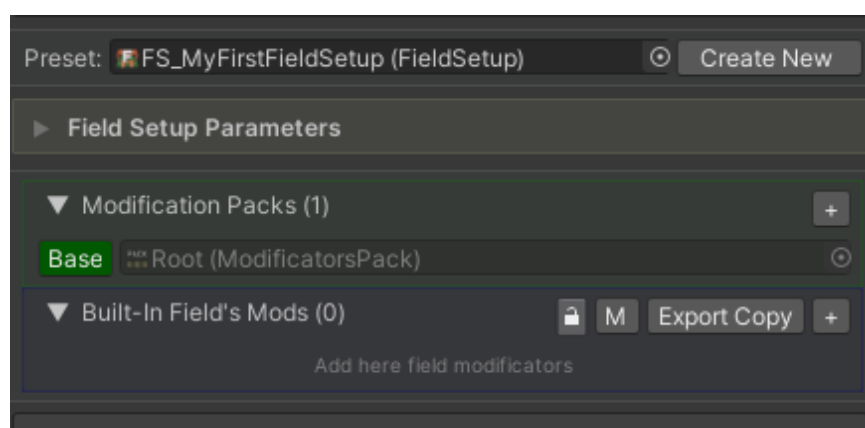
Now you can start adding Field Modificators by pressing the **"+"** button.
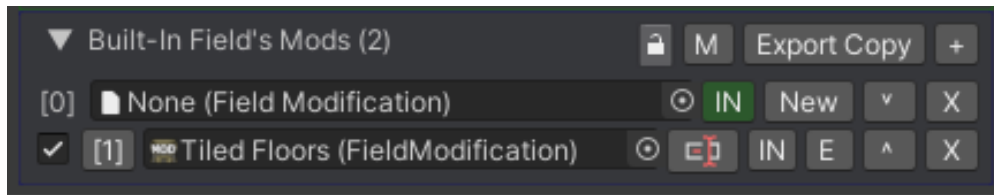
You can keep Field Modificators inside Modificators Pack (in root pack then also inside FieldSetup without creating additional file inside project directory) In most cases you will use **"IN"** button to create a new **FieldModificator** and put it automatically inside **ModificatorsPack**.
With toggle on the left you will quickly disable / enable Modificator, very helpful when designing rules.
W**ith "^" and "ᵛ" buttons you will change order of modificators which is very important**, since each modificator have only access to field state on modificators executed before it.
To know what other buttons are doing, just enter on them inside Unity Editor and tooltip will appear.

When you have modificators, just hit buttons on the left "[0]", "[1]" etc. to display Field Modification inside the inspector window.
Clicking on these buttons with the right mouse button will display additional options for duplicate modificator, export copy or variant.

2: Designing Your Field Setup

Your first field setup:

Open Field Designer Window and create a Field Setup preset using the "New" button.
Unfold "Built-in Field's Mods" to display Field's modificators list.
Hit "+" button to add a new field in list, hit "IN" to generate a new Modificator inside Field Setup's asset file.
Hit the "0" button or double click on the Modificator field to display it's options.

Field Spawners:

Inside the Field Modificator window you can assign prefabs for spawning.
Hit the "Add Spawner" button and select the prefab you want to generate.
Without any rule setted, the spawner will generate selected prefabs on every cell of the provided grid.
Now you can add rules to define how your objects should be generated.
Field spawners can have assigned tags for rules to detect wanted objects on cells, you can assign multiple tags by using "," commas like Prop,Lamp,OnCeiling. Tag assigned to FieldModificator will be applied to all spawners within.

Field Rules:

When you hit "Add Rule" button you will see different categories to choose from. You will find rules for moving/rotating spawned objects, for allowing to generate in certain conditions or more complex rules which are doing many things at once.
Each rule contains description which can be displayed when you **enter on it's header inside inspector window** (after adding it to spawner)

You can reverse/negate rule logics, which is pretty important, sometimes you might want to get opposite effect from some rule, then just click '#' button to negate rule logic. This feature is very useful and it's worth remembering.

You can do the same with tags, adding "!" before tag name will make rules affect every tag excluding ones with "!".
Example: "!Floor,!Ceiling" so all spawns without Floor and Ceiling tag will be gathered by rule algorithm.

Short description for each Rules Category:

<u>Full Transforming</u>: Rules which are changing every aspect of objects like position, rotation and scale. In future versions there will be more rules.
<u>Transforming</u>: Rules which change only position, rotation or scale.
<u>Quick Solutions</u>: More complex rules with algorithms which replaces usage of multiple other rules to get desired effect with just one rule.
<u>Placement</u>: Checking conditions related to cells placement on grid.
<u>Other</u>: Just helper decorators, not changing anything in grid.
<u>Modelling</u>: Rules related to changing generated objects appearance etc.
<u>Field And Grid</u>: Rules related to field variables, other fields, grid size etc.
<u>Count</u>: Rules which are limiting count of spawned objects.
<u>Collision</u>: Rules related with collision check (procedural generation grid is using mostly bounding box collision, no raycasting)
<u>Cells</u>: Rules which are modifying other cells, like removing spawns, preventing generating for selected ones etc.

Setting up props:

For a more practical presentation on how to set up props on your grid **check the tutorial videos on Youtube**. **(Tutorial links soon)**
Here are some tips: when you want to set some **debris** on the floor, use just randomized positioning with Transforming rules, separate spawns with Placement->Cell Distance rules using debris spawner name and in addition use Count->Spawning Probability to avoid too many objects. Also don't forget to set "Cell Check Mode" of spawner to random for additional randomization.
If you want to generate **wardrobes or other furniture**, use Full Transforming->Get Coordinates and gather coordinates from walls, adjust positioning and rotation plus add similar count limits / distance separation like with debris.
If you want to generate **groups of objects like a pile of card boxes**, prepare a "Multi Spawn Emitter" component and create a prefab out of it then spawn it in a similar way like wardrobes.
If you want to generate **multiple objects in one cell, like narrow shelves**, try using Quick Solutions->Sub Spawner to generate additional objects in the same cell but with other customized rules.

Complex Cases:

If you want to **create curved walls with your levels or for example floors with leaks on texture aligned to walls,** use the power of Quick Solutions->Wall Placer and Floor Placer components.
If you want to replace **walls with damaged versions every few wall segments**, you can use Full Transforming->Get Coordinates to align with other wall, then

Cells->Remove in position to remove previous wall and replace it with damaged version, then use similar conditions like debris in previous manual section, to spawn them every few segments.

In case you want to **create a giant underground tunnel and cover small objects with it,** you can try using a scaled grids feature.
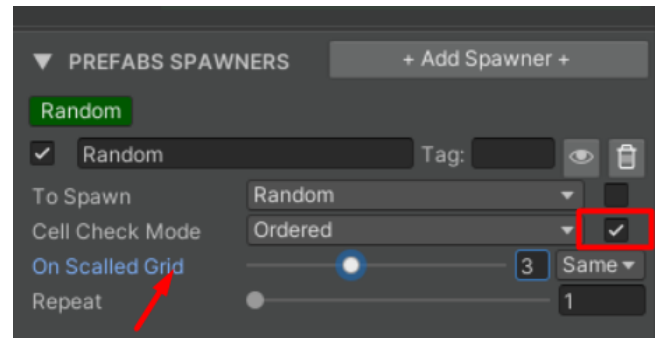
You can switch scaled grid for spawner with the toggle next to "Cell Check Mode" Use Grid Painting to see how it works and check **this tutorial video (link will be added soon)**.

Common Logic Rules Nodes:
Most commonly used nodes are:
Full Transforming -> Get Coordinates
Quick Solutions -> Wall Placer, Floor Placer
Placement -> Distance to other, Check Cell Neighbours, If Cell Contains Tag
Count -> Spawning Probability
Cells -> Remove in Position

Optimizing:
If your generating process takes too long, you can optimize it a bit with approach like follows:
Using "Cell Check Mode" called "Total Random" is not optimal, it's shuffling all grid cells, when you use Modificator on a large grid it's better to use other cell check modes.

It's best to set simplest logics as first in spawner, if the first simple check is not allowed to spawn then the rest logics are ignored and will not cause CPU overhead.

Simple logic nodes have "Lightweight" suffix in tooltip when you enter node's header title.

3: Using Many Field Setups

Build Plan:

Build Plan can be filled with customized settings for multiple Field Setups.
It can be used by other components to generate multiple grids.
You can prepare Field Setups for generating whole rooms like guest room, bathroom, kitchen etc. and use other components to generate them in the wanted setup.

Check example scenes of generating dungeon game, which is using build plan to generate key room, boss room etc. restrict their position to be near exit doors etc.
Also check this tutorial video for some more explanations. (tutorial link soon)

Example Generating Components:

Building Plan Generator (randomized generating): It's an old component which was made at the alpha stage of this package. It's not using the newest "CheckerField" framework but can be found useful for some users. It's trying to generate corridors and rooms out of the provided room inside the Build Plan preset.
Facility Generator (randomized generating): It's similar to BuildinPlanGenerator but uses a "CheckerField" framework which supports rotating non-rectangular room shapes to fit them to generated corridors and rooms.
Grid Painter (manual level design): It's component which allows to paint cells on scene, paint command guides very useful to set up doors between other Grid Painters.
MiniCityGenerator (randomized generating): Simple component to generate roads and buildings around them.
RectangleOfFieldsGenerator (controlled generating): Generates multiple FieldSetups within a custom rectangle shape, useful when generating house buildings with rooms inside them.
SimpleDungeonGenerator (randomized generating): Very similar to Building Plan Generator and Facility Generator but with some different rules.
SimpleFieldGenerator (randomized generating): Simple code example of generating objects with provided FieldSetup preset.

4: Additionals

Object Stamper:

Object Stamper is a component which can spawn objects on ground using raycasting with randomization. There can be random prefabs and each can have its own probability level to be chosen to spawn.

Add the ObjectsStampEmitter component and create a Stamper Preset using the "New" button, open the preset and add prefabs through the inspector window.

Adjust random position/rotation/scale settings and raycasting settings.

Check restrictions and overlapping configuration to spawn objects according to your needs. Now you can hit the "Randomize Preview" button to see ghost gizmos of target object to spawn, you can create object with this component as prefab and use with PGG or place around the scene to spawn randomized objects with raycasted placement.

Object Multi Emitter:

It's using an Object Stamper preset or multiple presets to spawn multiple objects in configured areas. It is really useful when you want to generate piles of objects or objects on wardrobes, tables.

It can make generated spaces much more fulfilled.

Pipe Generator:

Component which uses modules of any shape to generate path towards target or random position. Modules must be prepared, meaning must have defined points to which other modules can join with target out rotation at target point.

It can make generated spaces much more interesting.

Checker Field Framework (for programmers):

Checker Field classes are providing workflow for generating grids.

This framework is easy generating custom build plans by auto cell-path generating, quick cell collision checking, snapping cells or groups of cells from one to another.

Check example generator components commented code lines for more.

5: Advanced Specification:

Global Rules:

Some rules have option to be setted as global ("G" button)
When enabled, all spawners in Field Modificator will have this rule applied. Also when execution order matters, global rules setted lower will be queued to be executed after all other rules in spawner, global rules on the top will be queued to be executed before all other rules.
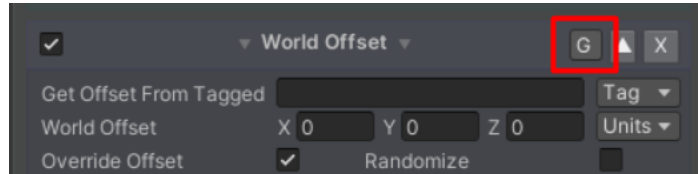
Field Setup Commands:

Field setup can have assigned Instruction Commands definitions which can be used by Grid Painter and custom coded grid generators to create door holes in walls by replacing wall model with doorhole wall model. You can also use commands to just spawn additional objects on the grid like painting fences etc. This Cell Instructions opens many new possibilities for designing your Field Setup.
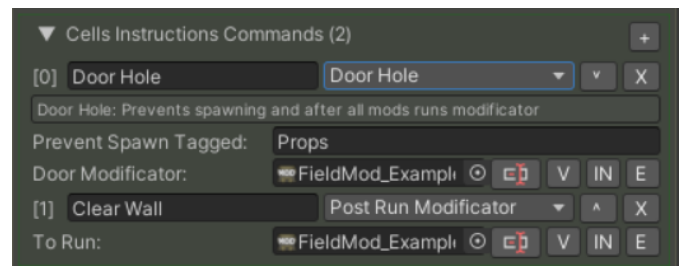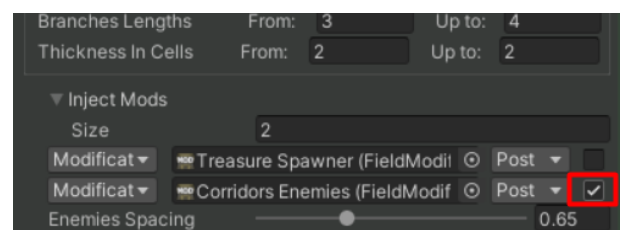
Field Setup Variables:

Field Variables can be used by spawn rules to change their parameters.
In most cases you would like to use field variables when injecting modificators to other field setups.

Injecting:

Injecting means running Field Modificator / Group of them as addition to Field Setup without adding them to the Field Setup preset file.

For example: you have Field Modificator which spawns enemies, you want to inject enemies spawner to some rooms, but in room A you want to have a lot of enemies and in room B just a few of them. You can use the Distance Check rule for setting spacing between enemies and assign to it Field Variable

(assigning? -> bottom of page 6) then override variables with the injecting field and change the field's variable value up or down. (To assign variable modificator must be child of Field Setup with needed variables)

6: Coding Custom Spawn Rules

If you want to code custom rule to be used in your Field Modificator Spawner, go to:
Right mouse button somewhere inside project directory -> Create -> Scroll and find **FImpossible Creations -> Procedural Generation -> Scripting Templates ->** Choose Template, name it and open automatically creates .cs file.
(If you use Assembly Definitions this file must be created in PGG's directory or directory with Assembly Definition referencing to PGG assembly)

Right away you can add your rule inside spawners under category "Custom"
You can change it by setting a different namespace for the class file.
Each spawn rule must have at least one visible public variable to avoid GUI errors!

Now depending on what your rule will do, you will have access to the current state of the grid, to the spawn data grid is attempting to generate, Field Setup file when you override right methods.

You should choose correct **EProcedureType** for your rule:
Procedure - Conditions + changing spawn properties
Rule - Conditions for spawning (override just CheckRuleOn())
Event - Just changing spawns properties (override just CellInfluence())
OnConditionsMet - Will be executed at the end only when all conditions are met
Coded - Running CheckRule and OnConditionsMet to give wide variety access

And override right methods:
CheckRuleOn() Executed on every cell until conditions are met
Use variable CellAllow to make rule allow or disallow spawn
CellInfluence() Executed only once when occupied cell is known after running all rule checks
OnConditionsMetAction() Executed at the end of all rules if all conditions met
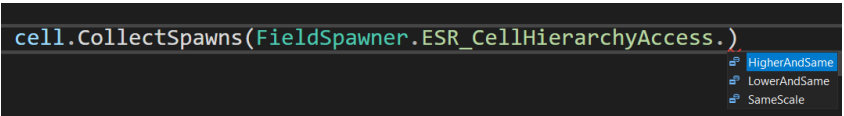
Check code of other rules to get acquainted with grid logic.
I suggest checking simpler ones at first, like SR_PushPosition, SR_SpawningPropability.

Optional: To support **accessing spawns of scaled grid's cells** you can use code like this:

```
cell.CollectSpawns(FieldSpawner.ESR_CellHierarchyAccess.)
```
HigherAndSame
LowerAndSame
SameScale

but if you don't need scaled grid support you can use method like:
cell.GetSpawnsJustInsideCell()
Optional: To support **Field Setup variables**, you must add
public **SpawnerVariableHelper** VariableHandlerName;   below your rule's variable, it
will create this half-dot on the left of the property inside the inspector window.
Then you can use it in code like: VariableHandlerName.**GetValue**(defaultValue);
to make an effect in the rule algorithm, but you need to do one more thing to make all
of that work.
You need to **override GetVariables()** method and return your variable handlers in it.
You can use quick method <u>if you need to use just one field variable handler</u> in your
rule: return VariableHandlerName.GetListedVariable();

Soon all mentioned tutorial links will be fulfilled!

In the future the manual might expand more!