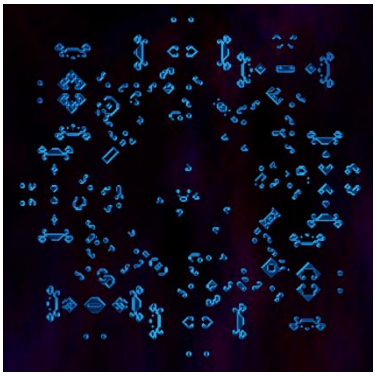
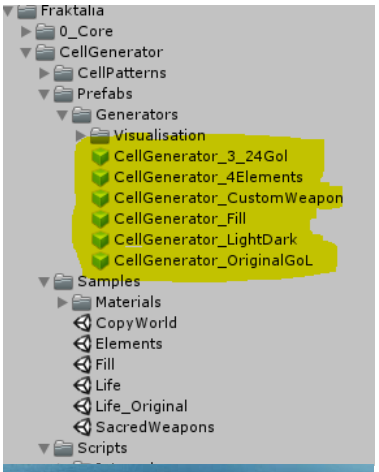
# CELL GENERATOR

## *INTRODUCTION:*



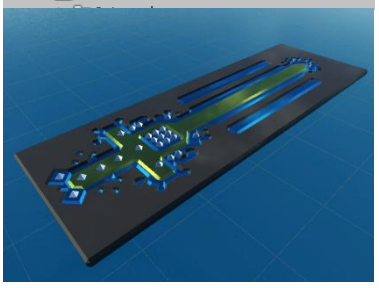
The cell generator is a system which uses cellular automata to create objects procedurally according to cells which are affected by defined rules. The most famous cellular automata system is Convay's Game of Life which uses 2 simple rules for cell resurrection and death. These rules are enough to create an unpredictable outcome. Also the cell generator has properties of a voxel engine but without memory saving data structures because this generator is optimized for fast cell updating. However it can also be used for 2D voxel maps or 3D maps with very low resolution so it does not eat up your memory. This generator also contains a system where custom rules like flood fill or element cycles (water > fire > air > earth > water) can be defined. Also the whole generator is burst optimized so it is recommended to install the burst package for performance boost.

## *HOW TO USE:*



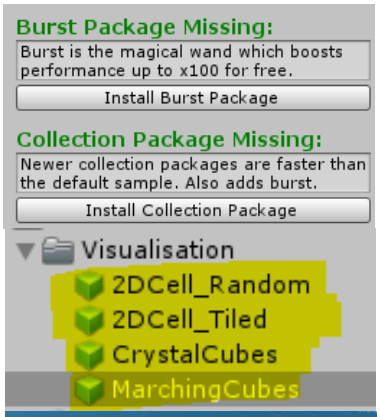

For fast testing, just drag in one of the provided CellGenerator prefabs in your scene. All CellGenerator prefabs have visualizations attached so the initial system becomes visible immediately.

The prefab contains the CellGenerator, CellSaveSystem, a CellModifier component which allows you to modify the block by holding CTRL + Left/Right/Middle mouse click. The main settings for the CellGenerator are the Boundary which defines length, width and height and the Root Size which defines the volumetric size of each cell. Usually the Y value is 1 or 2 since 2D cell fields are recommended. If the Y value is lager, the CellGenerator basically turns into a 3D voxel engine with cubic memory requirement increase.

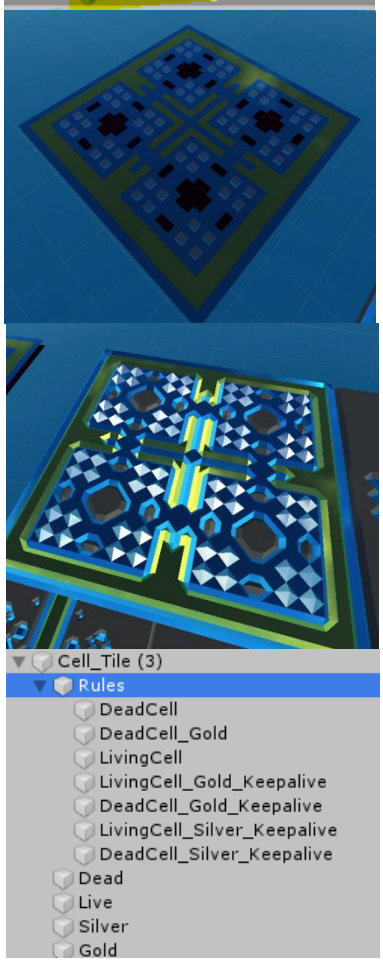


The CellSaveSystem provides options for saving the volumetric dataset and automatically saves the cell data when saving the scene itself. The standard setting "Scene" is the most convenient method as it simply stores the data into the ".unity" scene file and is fine for 2D cell maps with smaller resolution. However the Unity serialization slows down the Unity Inspector when the size of the cell data becomes too large especially if it goes into the third dimension due to cubic memory increase. For such cases, it is better to save it as ScriptableObject in order to keep the Unity scene file small.

When importing this asset the first time in a fresh Unity project, the **Burst and Collection** package probably is not installed. Fortunately it is now easy to install the latest version by clicking on both buttons which are visible at the CellGenerator component. Burst will boost the performance and Collections updates the internal data structure which also boosts performance.




The visualization itself is separated from the main CellGenerator GameObject and is a separate hull generator attached as child to the CellGenerator. Each CellGenerator can have as many hull generators attached as desired. Most hull generators listen to exactly one cell value which can be between 0 and 255. The most commonly used hull generators are MarchingCubes and Crystallic but more may be included in future updates. All generated meshes can have collision detection for GameObject interaction. The cell data of the 2 right images is identically but different hull generators are used for the visual representation.




When the CellGenerator inside the editor is selected, you can modify the block by holding CTRL + Left/Right/Middle click whenever a CellModifier component is attached (which is usually the case unless removed). Saving the scene will automatically save modified CellGenerators. On scene initialization, the Cell data is regenerated by the CellSaveSystem or if disabled, can be loaded dynamically later.

In order to simulate Cellular Automata, a game object with a various rules is attached to the generator. The rules are fetched on initialization and converted into a native format. This allows the creation of Cellular Automata rule without touch burst compiled code.

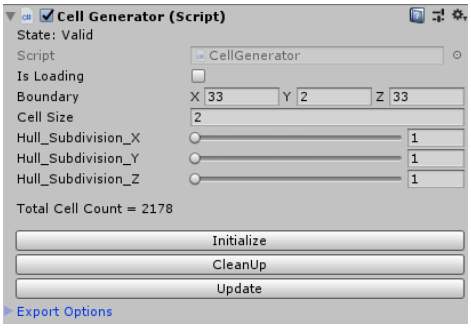
Since the CellGenerator is supposed for 2D cell fields, a variety of modifying tools are included which can be used as brush by the CellModifier component. It is also possible to use .rle cell pattern files which can be downloaded on the official game of life wiki page:

https://www.conwaylife.com/wiki/Main_Page

# MAIN COMPONENTS:

The main components are the CellGenerator itself, a variety of CellRules to define rules, a variety of hull generators for visualization and supportive components such as the CellModifier and CellSaveSystem. Each CellGenerator could exist completely alone without any supportive components but then have limited functionality as it only provides the data management.

## CELL GENERATOR




The CellGenerator is the main component of the cellular automation system and provides the internal data management and applies provided rules whenever an iteration cycle is executed. The CellGenerator contains parameters about the boundary and the hull generation process:

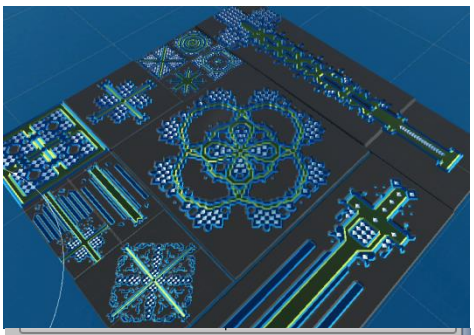
**Boundary:** Describes the size of the cell field and only allows positive values. In most systems a 2D field is desired and it is recommended to keep the Y value low since the Y axis is defined as "Height" in Unity3D. Therefore brushes and tools expect the Y value to be the height. The Cell Count label shows the amount of cells the field will contain (X * Y * Z). The safety limit is set to 5000000 cells. Higher Y values would turn the cell generator into a low resolution voxel engine.

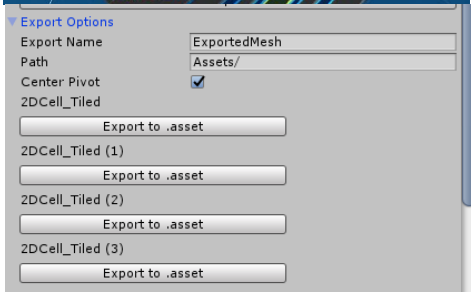
**Hull_Subdivision X, Y, Z:** Defines the subdivision of hull generators. Higher subdivision counts are recommended for large cell maps in order to reduce computation costs for the hull generators. Only those regions where cells have changed are updated.

**Cell Size:** Determines the physical size of each cell. A blue boundary box indicates the total space a generator will occupy.

**State:** Valid if initialize and no errors, Invalid if not initialized or errors are detected.

**Initialize**: Initializes the cell field.
**CleanUp**: Clears the cell field.
**Update:** Execute one cell iteration cycle.






The CellGenerator also include an export functionality which automatically detects attached hull generators. When exporting, all subdivided regions of a hull generator are merged together to generate one asset mesh file which is then stored in your asset database.

## TECHNICAL PART:

### FUNCTIONS:

**Initialize():** Initializes the cell field. Also has a dynamic version which is used by the save system to prevent performance drop during initialization.

**SetCells()**: Provides a variety of options to modify cells directly.

**WorldToCellPos/LocalToCellPos/CellToLocal/CellToWorldPos:** Simple coordinate conversion functions.

**SetVisualisationDirty():** Marks the generator as dirty and launches visualization updates.

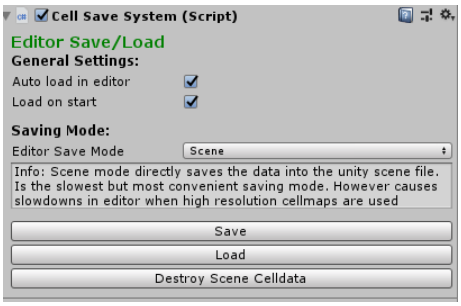
**Cleanup():** Destroys the Cell Field

### PROPERTIES:

**IsInitialized:** Property which is true, if the is generated.

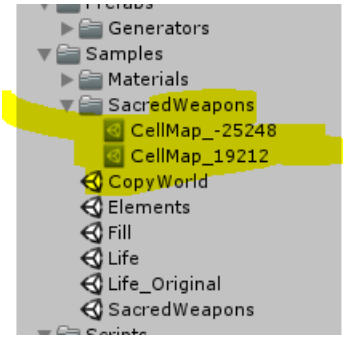
**IsLoading:** Returns true if loading processes by the CellSaveSystem are going on.

## CELL SAVE SYSTEM:




The CellSaveSystem is the main component for persistence and should always be together with the CellGenerator. The only exception occurs if the content is completely procedural and persistence is never intended. This component is also required since modifications inside the editor should be saved. If the CellGenerator was modified in during edit mode, it will be marked dirty and saves the data automatically when saving the scene. (pressing ctrl+s)

There are 3 different saving options where the first one is “Scene” which saves the cell data directly into the scene. This saving method is fast and convenient since there are no extra files to manage. However this mode will slow down the inspector when high resolution data is used. Since the memory demand of Cell Generators is low, “Scene” saving method is recommended.




The second mode is “Scriptable Object” which stores the data as scriptable object in the asset database. This option causes less overhead for Unity and keeps the scene file small. Also causes no serialization overhead which is useful for high resolution maps. Also this method is better for version control systems such as git since there are more small files than one big scene file. Also merge conflicts are easier to avoid since data can be merged while Unity scene files still have merge issues in git. Also data from the “Scene” option is removed when saving with this mode since having data in the scene file is obsolete when it already exists as scriptable object. Additionally, this saving option comes with a management system and has following settings:

- **Duplicate Map on Clone:** If true, duplicates the data if you duplicate the generator inside the editor. It clones the original data and automatically assigns it to the cloned object. The naming is “CellMap” + the instance id of the cloned GameObject. The location of the cloned map is in the same folder as the original map.

- **Save in scene sub folder:** If true, a new cell map is generated and placed into a subfolder where the Unity scene is located during saving if no map is assigned. The subfolder is created automatically if it doesn't exist yet. If false, newly generated voxel maps are placed into the assets folder.
- **Remove cell map on delete:** If true, cell map is automatically removed from the asset database. Note: Removing is not undoable and is permanent!

The third mode is "Persistence Datapath" and just requires a name as identifier (without ".CELL" ending). It directly stores the cell map as .CELL file into the persistent data path of your target device. This mode is supposed for persistence during gameplay where the player can save cell maps. Savings are not handled inside version control systems since these files are probably not inside the repository and will only exist locally unless shared by external services. This saving mode is the fastest for saving and loading and is optimized for real time saving/loading.

If **Auto Load In Editor** is true, cell maps will be loaded automatically when entering edit mode or loading the scene.

If **Load On Start** is true, the cell map will be loaded on scene initialization during play mode.

## TECHNICAL PART:

### FUNCTIONS:

**Load():** Loads the cell data and initializes the CellGenerator

**SaveBinary():** Saves the cell map as binary file. FileName member variable is used as identifier for example "CellMap1" without ".VOXEL" as ending.

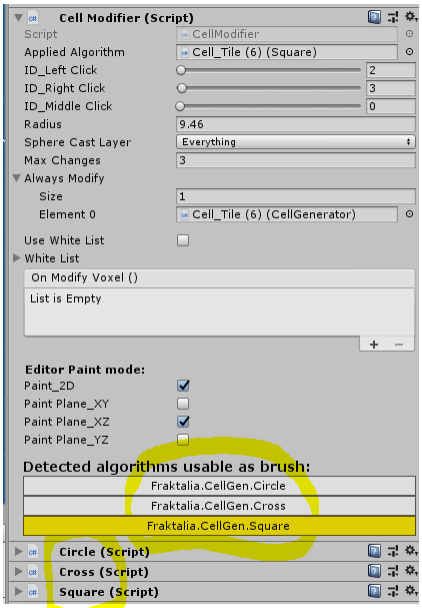
**RemoveBinary():** Removes the binary file at persistent data path. FileName is filename.

**DynamicSave():** Real time saving option which does not cause performance spikes. Saves the cell map as binary file into the persistent data path and should be called without interrupting gameplay.

**DynamicLoad():** Real time method to load any cell map from any source without interrupting gameplay. CellGenerator can be modified during loading. Especially useful in open world scenarios where stuff is loaded when the player near a specific region.

# CELL MODIFIER

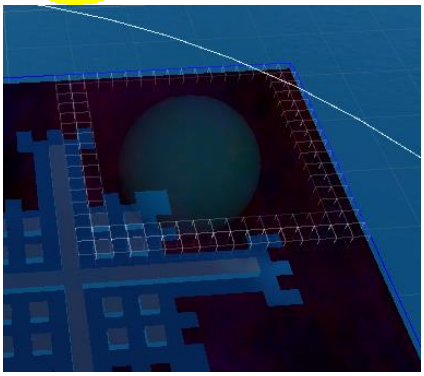
The Cell Modifier is the main tool to modify a cell field inside the editor. This component should exist on the same GameObject containing the generator but it can also exist alone. If alone, the Cell Modifier uses collision detection to find the target CellGenerators.




In order to paint, have the GameObject with this component selecting and hold CRTL (STRG) + Left, Right or Middle mouse click. When holding CRTL, a sphere or brush will indicate the target paint location.

Also cell pattern algorithms can be used to paint with brushes. When a cell pattern algorithm is attached, buttons appear on the inspector where you can select the algorithm you want to paint with.

Left click on one button selects the algorithm and assigns it to the applied algorithm variable (manual assignment is possible but using the buttons is more convenient). Right click on the selected(yellow) algorithm button will deselect it.



**ID_Left, Right, Middle:** Value to set when painting.

**Radius:** Describes the radius of the modification.

**Sphere Cast Layer:** Layer for automatic generator detection. Only used when no brush is selected.

**Always Modify:** List of generators which should always be affected. Used for painting on more than one generator.

**Use Whitelist:** When true, only generators inside the whitelist will be affected.

# MODIFING ALGORITHMS

Modifing algorithms are components which generate a defined pattern which can then be used to modify the cell field either directly by clicking on the apply single/full button or as painting brush using the Cell Modifier.

**Target ID** is the main parameter and describes which value affected cells will obtain when applied. This is overridden by the Cell Modifier when used as brush.

**Start/End Index** are used to define the points calculated and Index shows the current step when single steps are applied instead of fully complete steps.

**The Origin and Z Position** describe the starting point where Z Position actually is the height (Y axis)

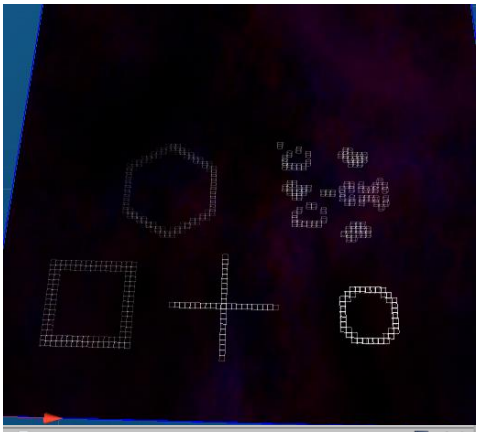
These components should be attached to the GameObject containing the Cell Generator as it fetches the generator automatically. Else you have to assign it manually.

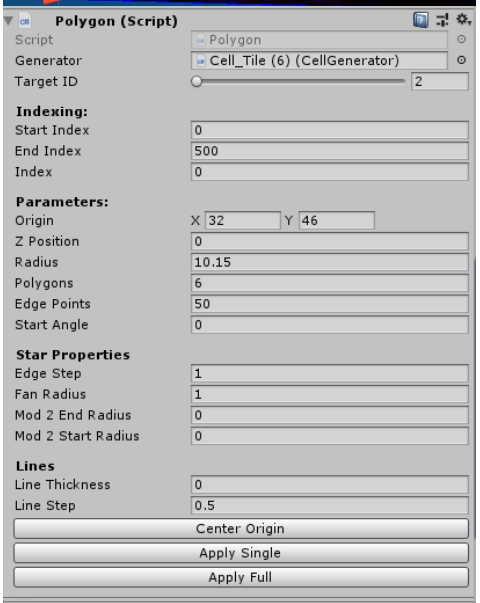
Also a preview will show where cells would be modified and is updated whenever parameters are modified. Common parameters are Radius, Steps, Line Counts, Circle Counts and are self-explanatory.

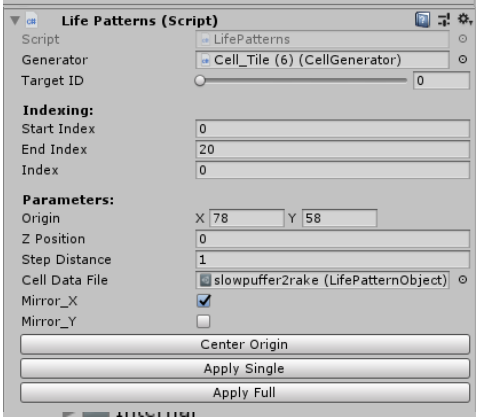
However the LifePatterns algorithm works different as it requires a Cell Data File. This file is an imported .rle file which contains information about a very specific pattern. These patterns are supposed to be used with the original Game of Life rules and can be obtained from the dedicated wiki page (1500+ files). The included samples are also from the large package on the wiki page.

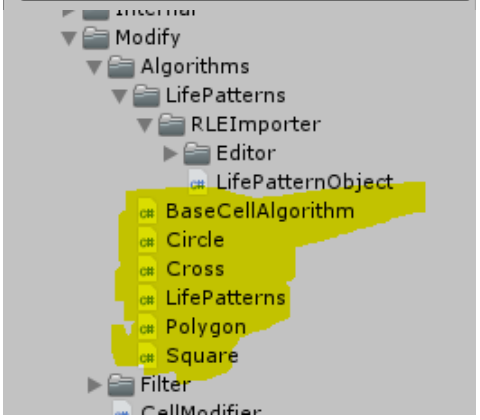
https://www.conwaylife.com/wiki/Main_Page

This algorithm ignores the Indexing parameters since no mathematical model is used to represent the pattern. Only the most basic parameters like Origin, Z Position and Step Distance (should be 1 for life patterns) exist. With Mirror X/Y you can mirror the pattern as it would still keep their simulation properties.

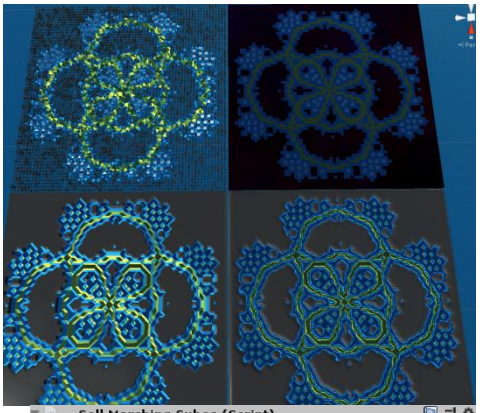


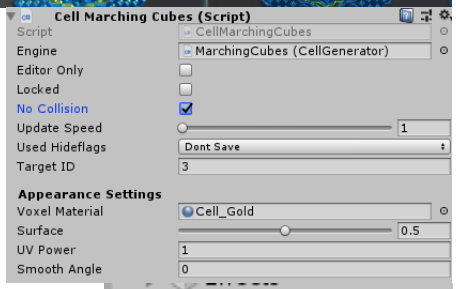


# HULL GENERATORS







Hull generators are responsible for the visualization of the cell map and must be attached as children to the CellGenerator. Each hull generator has its own settings such as material and any number of hull generators can be attached to the CellGenerator. The cell generation systems only use 4 of 256 possible states and are Dead(0), Life(1), Silver(2) and Gold(3). Therefore the generators have 4 hull generators assigned where each hull visualizes one state.

The first parameters which can be found on every hull generator is the **Engine** itself which is fetched automatically when attached to the CellGenerator.

The **Editor Only** flag can be set if the according hull should only visualize during edit mode as helper tool.

Any hull generator can also be **Locked** which prevents the hull from updating. It will continue immediately if locked is unchecked.

**NoCollision** as the name suggest disables mesh collider generation and usually is set. Uncheck this setting if you really need collision detection since this has significant performance impact.

Also every generator (future may have exceptions) have an **Update Speed** parameter where the region updating speed is adjusted. This has no impact if no region subdivision is applied (Hull Subdivision X, Y and Z set to 1 on Cell Generator)

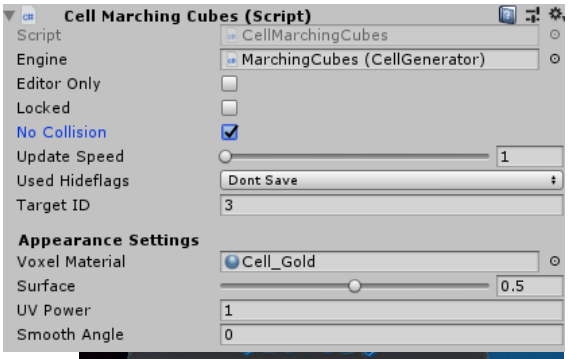
**Applied Hideflags** are set to "Hidden Don't Save" by default which hides generated GameObjects and prevents them from being saved into the Unity scene file. Saving the hulls is not necessary because it can be reconstructed at any time by the VoxelGenerator which also applies inside edid mode. Therefore saving generated hulls is obsolete and only would bloat up the scene file. Other options are "Normal", "Don't Save" and Hidden which are all combinations of Visible and being saved.

The most important setting is the **Target ID** which defines which Cell Value it should visualize. The hull generator in the image above wants to visualize "Gold" and therefore the **Target Id** is set to 3. It is also possible that future

Hull generators can also use MeshPieceAttachments which are

# MARCHING CUBES

Marching cubes is the most common visualization method for data and is mostly used for terrain or other natural environments because the appearance is completely smooth. The smoothness is either 0 or 1 or “Non-Solid” or “Solid”. If the cell value matches the TargetID, the cell is marked as “Solid”.

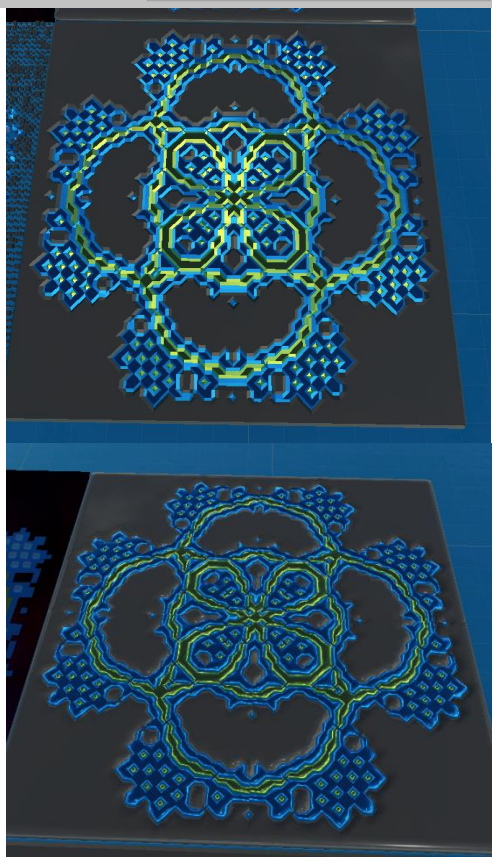



The appearance settings define the visual appearance of a hull generator. It usually contains a material, UV power and optional smoothing of normal coordinates.

**VoxelMaterial:** Material used for the mesh pieces.

**UV Power:** UV Multiplier for the mesh pieces

**Smooth Angle:** Smoothing Angle of normal coordinates. Note: If the smoothing angle is greater than 0, an extra computation step is required which increases higher performance impact. The middle image show the result with smoothing angle set to 0 and the bottom image has a smoothing angle of 180.



# CRYSTAL

This hull generator generates a crystalline structure and can be used for 2D only visualizations by using the default quad provided by Unity as Crystal mesh (top image). This generator also has the resolution settings which are also found in marching cubes hull generator. The only difference between crystal and marching cubes are the crystal shape settings:

**Crystal Mesh:** Defined crystal shape. Low poly shapes recommended.

**Voxel Material:** Material applied.

**Offset Min/Max**: Positional offset.

**Scale Min/Max**: Random scaling of each crystal

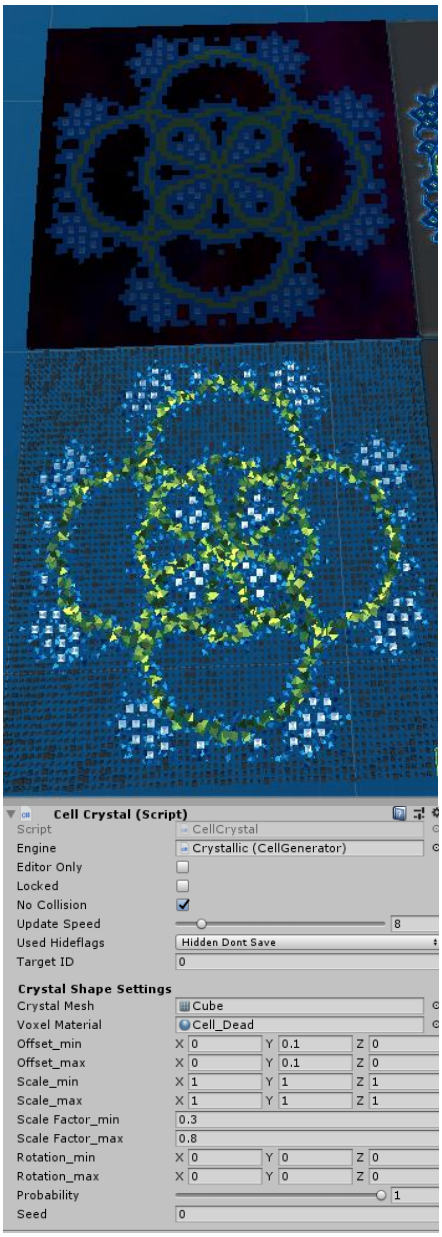
**Scale Factor Min/Max**: Multiplier.

**Rotation Min/Max**: Random rotation of each crystal

**Probability:** Probability of crystals being generated for each solid cell.

**Seed:** Seed used for randomness.

The performance impact of the crystal hull is lower than marching cubes especially if the crystal shape is the default quad. Therefore this visualization is recommended when the cell field is huge. Also randomness can be used to give the result a more interesting variation like in the middle image.

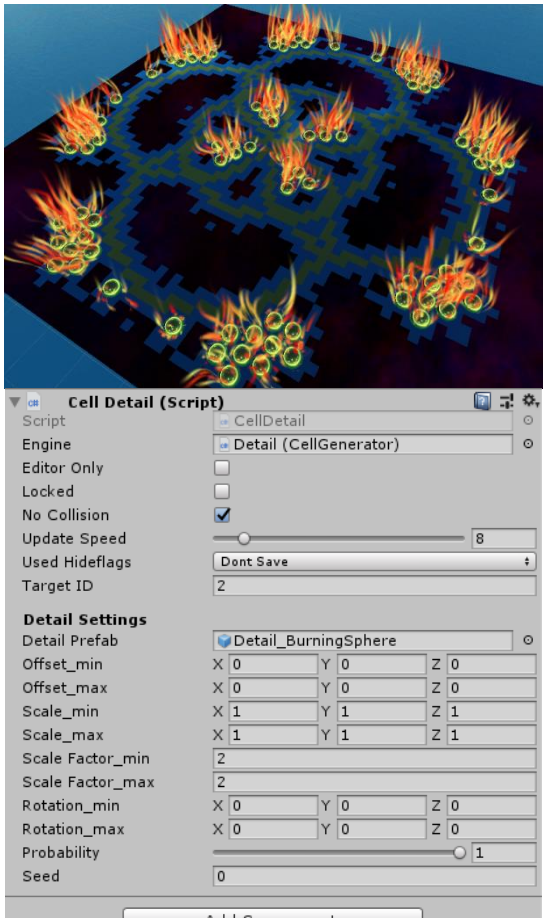


# CELL DETAIL

The Cell Detail hull generator is similar to the crystal one except that it instantiates a prefab for each cell it wants to visualize. Also the settings are almost identically to the crystal hull generator. The only difference is that you have to assign a GameObject instead of a mesh shape.

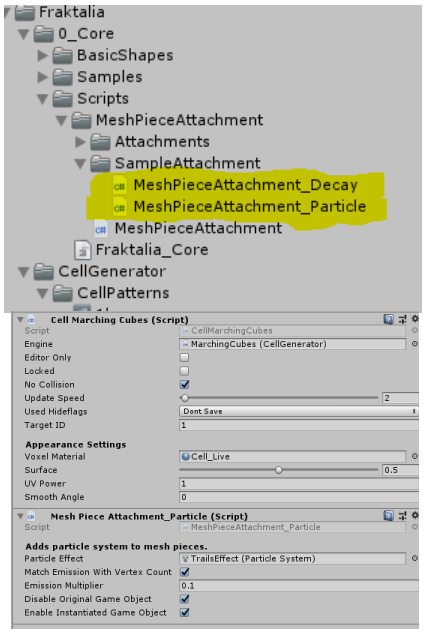
Since it instantiates a GameObject for each cell it matches, it is recommended to use this hull generators for rare cell ids since it is supposed to be used for details. If the visualization only contains a mesh and some effects, it is better to use the crystal hull generator instead.

The only reason to use this hull is when the cell should somehow interact with the outside world (not cell system world)



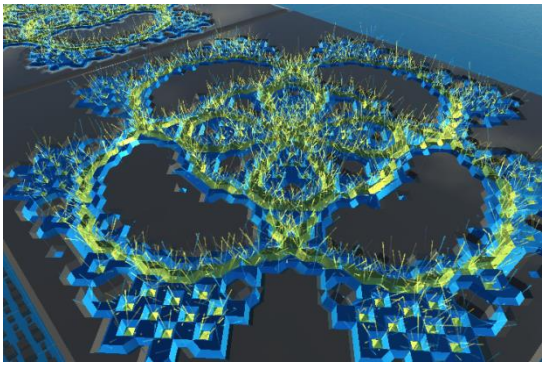
# MESH PIECE ATTACHMENTS

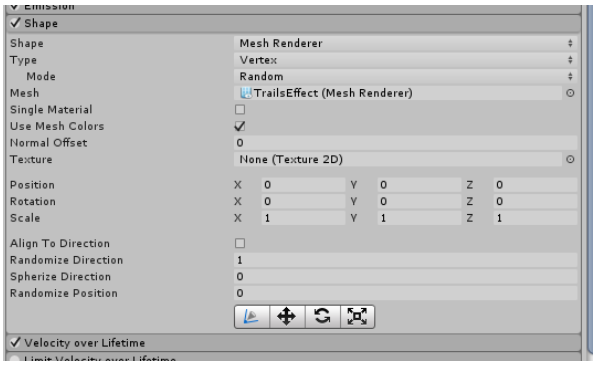
Every hull generator which generates mesh pieces can be further modified by applying mesh piece attachments. These scripts are located inside the 0_Core folder which is a special folder shared between multiple Unity Assets and is updated regularly. In order to use an attachment, simply add the attachment as component to the GameObject containing the hull generator which should be modifier.




The sample attachment below instantiates an assigned particle system and attaches it to every mesh piece. Then it also applies the procedurally generated mesh as shape to the particle system. Additionally it also controls the particle emission in order to maintain a stable particle effect.

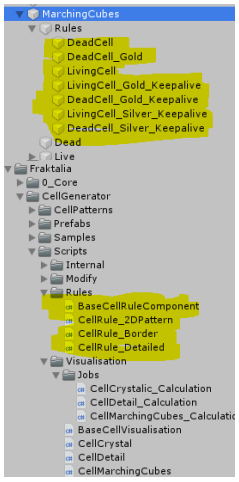
The image below shows the result with particle effects. The shape module of the original particle system must be active and the shape set to “Mesh Renderer”.





# CELL RULE SYSTEM

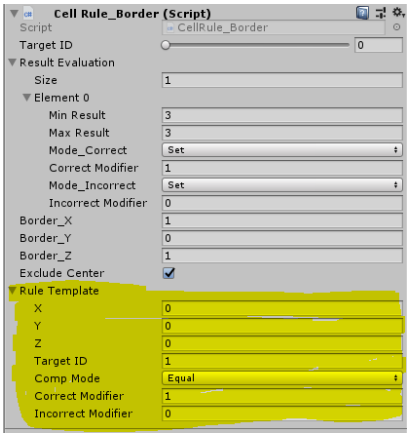
The cell rule system is used to implement cellular automata functionality like Convay's Game of Life. The cell rule system is implemented by simply attach GameObjects which contain a CellRule component to the CellGenerator. It is also allowed to group the rules in an empty GameObject as it is done in the samples.




The basic principle of the cell rule checks the ID of each cell and applies one or more rules defined for the specific ID. Therefore every CellRule component has a **TargetID** parameter. The rule appliance algorithm is a 2 step process. The first step checks neighbored cells accordingt to rule templates in order to generate a result-value. Then Result Evaluation templates define what should happen to the checked cell according to the result-value calculated in the previous step.

# RULE TEMPLATE

Every rule template checks exactly one cell and and modifies the result-value according to the comparison mode. If the cell matches the comparison, the **CorrectModifier** parameter is added to the result-value. Else the **IncorrectModifier** parameter is added to the result-value instead.




The **X, Y and Z** value defines the neighbor of the cell which is currently checked by the rule system. So a value of 0,0,0 would check the center cell itself.

The Comparison Modes supports **Equal, NotEqual, Greater and Smaller**.

Every CellRule component has at least one Rule Template to set. Then according which derivate is used, more Rule Templates are generated procedurally. For example the CellRule_Border automatically duplicates the RuleTemplate to match the field defined by the Border X,Y,Z values. Otherwise you would have to define 8 or more cells manually which is tedious.

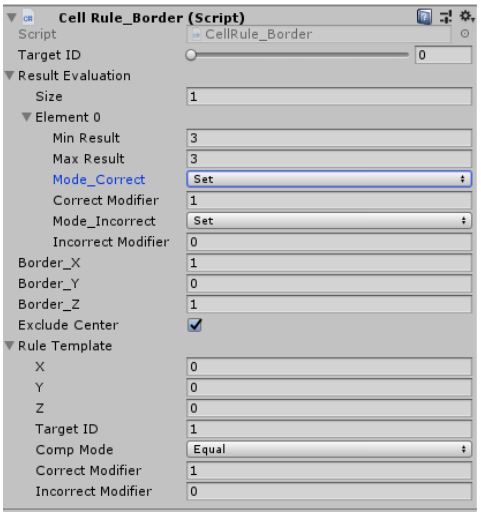
# RESULT EVALUATION

After the result-value has been calculated, result evaluation templates decide what will happen to the cell being checked according to the result-value. A cell rule can have multiple Result Evaluations and if multiple evaluations would apply the correct modifier, the latest will have priority. The correct modifier is applied if the result-value is between the **Min Result and Max Result** parameter; else the incorrect modifier is applied. The possible Modes to apply the modifiers are Set, Add, Multiply and Divide but mostly only Set and Add are required. If mode is set to add and the modifier is 0, the ID of the target cell will not be modified as adding 0 to something has no impact.

# CELL RULE COMPONENTS

## CELL RULE BORDER

The Cell Rule border is a convenient system to check all neighbors around a cell. The Border X,Y, Z value defines the border and it also can go into the third dimension. It creates multiple rule templates according to the original **rule template** defined in the inspector.




If border X and Z is 1 and Exclude Center is set, the rule will check its eight neighbors.

If border X and Z is 2 and Exclude Center is set, 24 cells will be checked.

The X, Y, Z values of the original rule template are used as offset and usually are 0.

## CELL RULE PATTERN

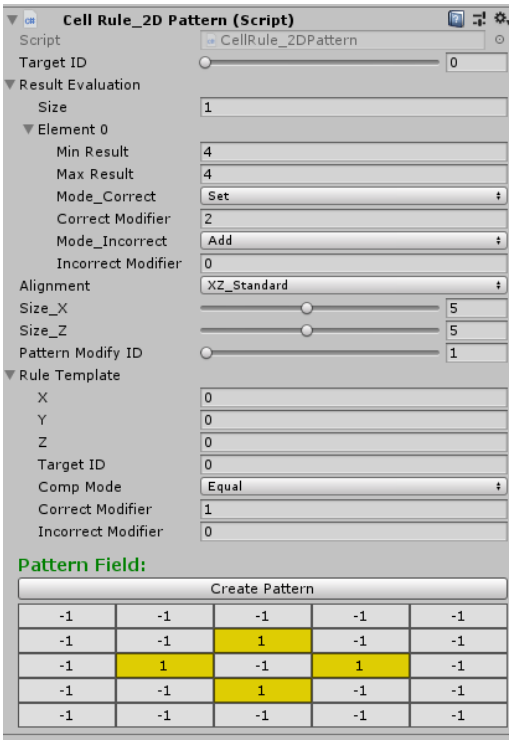
This Cell Rule component offers a convenient way to design rule templates according to a pattern field.

**Size_X and Size_Y** define the size of the pattern field and the field in the inspector adapts according to these values.

Each cell in the pattern field represents one Rule Template and when left clicking on one cell, the value is modified according to the **Pattern Modify ID** parameter and the color will turn to yellow since it is now enabled.

Right clicking on a cell will turn the value to “-1” which marks the pattern cell as disabled and the color will turn to light grey.

With this tool, it is possible to create complex requirements. The Alignment parameter is just the coordinate axis used and by default is set to XZ since the Y axis is the height.




## CELL RULE DETAIL

This is the most basic Cell Rule component and only contains a list of Rule Templates and a list of Result Evaluation templates. This component has the highest amount of flexibility but every rule has to be defined manualy.

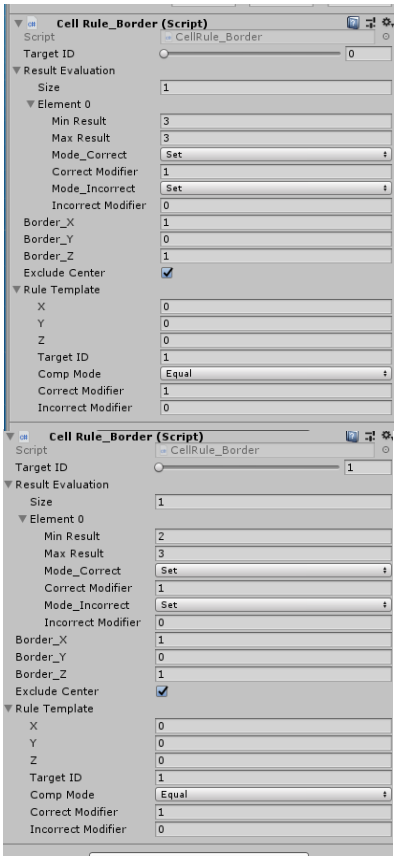
## EXAMPLE RULE: GAME OF LIFE

According to the original rules, the game of life has basicly 2 states which are “Dead” or “Alive”. A cell will survive if 2 or 3 neighbor cells are also alive, else the cell will die. A dead cell is ressurected if the eighth neighbor cells have exactly 3 alive cells.

In order to implement this, just two Cell Rule components are needed. The first rule is applied to dead cells (ID 0) as the Target ID is 0. This component will generate 8 rule templates because Border X and Y is one, excluding the center.

All cell rules increase the result-value by one if the ID is 1 or “alive” because the Target ID of the original rule template is 1. The result-value does not change if the check is incorrect since the incorrect modifier is 0. If the result-value is 3, the Result evaluation will set the ID of the evaluated cell to 1 resurrecting it.

The second rule is simply there to turn an alive cell into dead cells if the result-evaluation is not between 2 or 3.



## MOBILE SUPPORT:

Cell Generators can be used on any mobile device. However it is recommended to use lower resolution cell maps since mobile devices are not as powerful as modern gaming computers. Large Cell maps are no problem for mobile devices as long as no excessive visualization is used. The main limitation is GPU based because the CellGenerator is optimized for multi core CPU usage since the GPU already is busy rendering the geometry. Also having the burst package installed is highly recommended.

## VR/HIGH END SUPPORT:

Cell Generators can be used in VR applications as the hull generators only generate meshes procedurally. All my assets avoid the usage of any gimmicky stuff as the result is always a procedurally generated mesh which is rendered using the standard mesh renderer and a material of your choice.

For high end stuff, only the memory is the limit. Actually the maximum cell count is set to 5 million cells which should be high enough. Also large maps should have a higher hull subdivision as only a fraction of the region must be updated when cells change.

# FREQUENTLY ASKED QUESTIONS:

**Q: Why does a CellGenerator has boundaries?**

The underlying data structure requires a cell field which must be represented somehow. The size of a cell field is fixed after initialization in order to maintain consistency.

**Q: Does this asset work on lower Unity versions?**

The minimum required version to officially get this asset is 2019.2.11f1. It compatibility may work below 2019 until it hits the hard bottom of 2018.1. Below the asset will not work because the burst compilation does not exist below 2018.1

**Q: What are the best settings?**

For Mobile, a cell dimension below 256x2x256 and a hull subdivision of X4, Y1, Z4 is recommended.

Lower hull subdivisions means more vertices per mesh piece which increases GPU efficiency but increases CPU load during generation. Higher hull subdivision values reduce CPU load but increases the GPU load because more GameObjects are used.

**Q: Why is 3D cell generation not recommended?**

The cell generator is optimized for fast cell access in order to allow real time cellular automata functionality. Therefore the data structure has no memory optimization. Furthermore going into the third dimension will generate a cubic performance increase for example a field of 256 x 1 x 256 has 65536 cells. If we go into the third dimension such as 256*256*256, this turns into 17m cells filling up your memory. Cubic fields of lower dimensions are still possible like 128*128*128 which has a cell count of 2 million which is lower than the safety limit of 5 million.

## *LAST NOTES:*

If you have any questions, suggestions, bug reporting don't hesitate to contact me. If you are going to sell a game which uses this asset, inform me because I may buy your game and play it ☺

Contact Information:

E-Mail: m.hartl@fraktalia.org

Homepage: http://fraktalia.org/