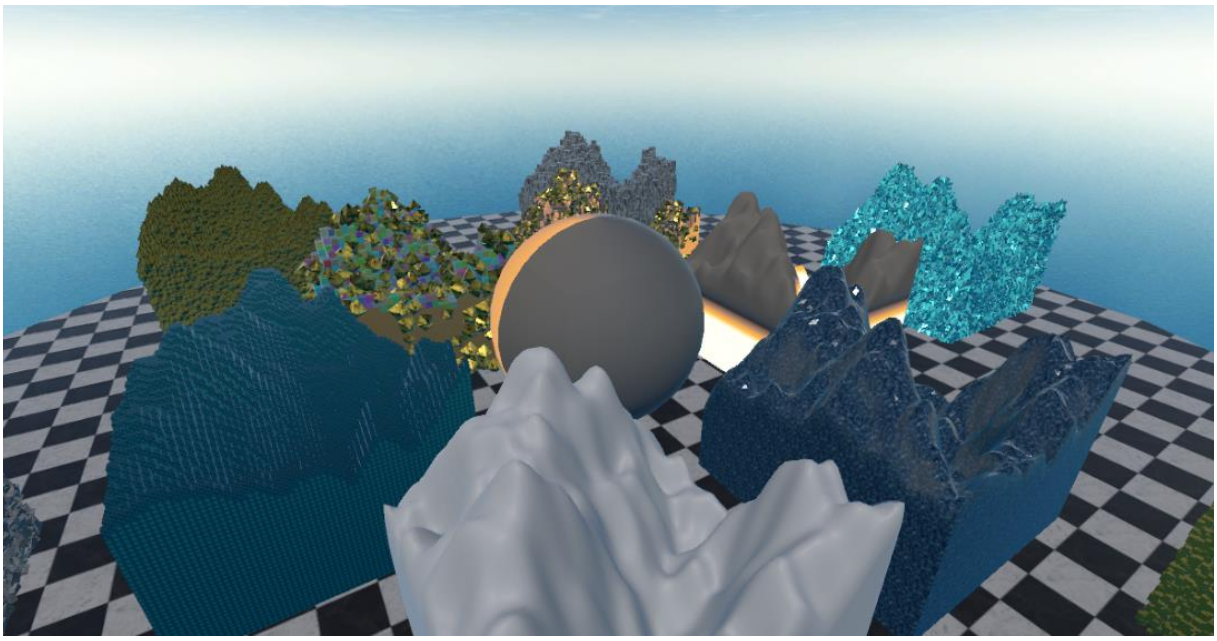
VOXELICA

INTRODUCTION:



Voxels are volumetric structures which are used to simulate real world atoms. Each voxel simulates one atom which can be modified. Voxel engines are often used in computer tomography and games such as Minecraft and Space Engineers. Often these games completely optimized towards the voxel engine which limits the usability by an great amount,

The aim of this asset is to provide a way to use voxels as an additional feature in your game while maintaining a normal workflow. Additionally, this asset also offers full flexibility over the data structure itself as nothing is hidden behind DLL walls or C++ bridges as it is often found in other assets since burst compilation makes this obsolete.

COMPATIBILITY INFORMATION

The voxel generator is fully compatible on every build target as it works fully independent of any unity function with the exception of burst and job system. The burst and job system is not mandatory but increases performance by a huge amount. The generator works on build targets without burst support but without the performance boost. Here is a list of tested build platforms:

- **Windows/Mac/Linux:** Normal and supports burst.
- **Android/Ios:** Normal and supports burst. Beware the lower memory on mobile devices!
- **WebGL:** Works but no burst support (may change by Unity in the future)
- **Consoles:** I don't own dev kits so I cannot guarantee functionality. Expected to work but burst may not be supported.

Technology Compatibility:

- **AR:** The voxel block can be placed as AR augmentation into a mobile app and can be modified after that. Keep in mind the reduced amount of memory on mobile devices.
- **VR:** Full VR support as the engine was implemented for a VR sculpting application.

Limitations:

- **32 Bit systems**: Memory limit to 4096mb RAM could be exceeded easily which results to app crash. Burst version 1.5.0 – 1.5.4 crashes the app on 32 bit builds for unknown reasons. ! (Engine warns you!)

RENDER PIPELINE INFORMATION

The Voxel Generator works independent of the render pipeline used. The important change is related to the material which has to be used when multi material setups should be used.

Almost all samples use the built in standard shader. The only exception are the materials which use texture arrays in order to smoothly blend between materials.

The texture array related shaders are used across multiple assets and are inside the 0_Core folder. This folder now has a HDRP section, containing a “TextureArray_RP.unitypackage” file which contains shader for the dedicated functionality.

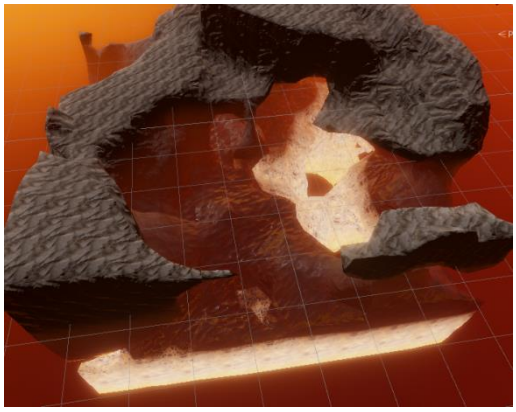
You have to either have HDRP or URP applied correctly in order to use those shaders. Double click to the TextureArray_RP file. After unpacking, the new content contains the .shadergraph files for HDRP and URP. Additionally a test scene is also included

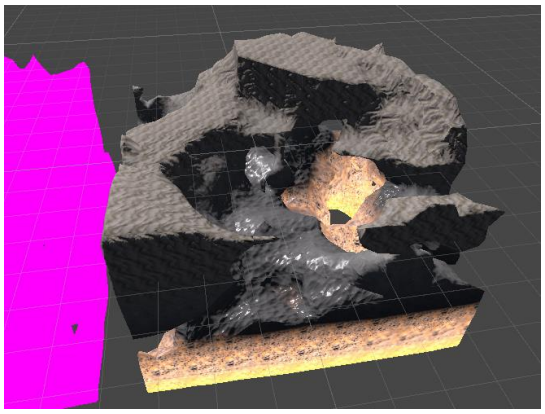
The sample scene contains one object representing the HDRP version and one showcasing the URP version. Since you cannot have both render pipelines active, the unused version shows the error like in the image below. The left one is the HDRP version and the right one shows the URP version.

The shader graphs itself are almost identical but minor things are still different which is the reason why separate versions are required.

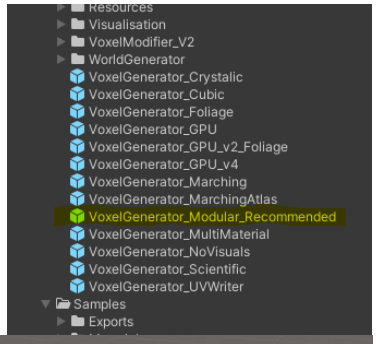
Since adding a UnityPackage into the asset causes the “Asset Validator” to fail leading to an instant rejection of the Asset by the review system, the HDRP/URP package can be downloaded from the here;

https://fraktalia.org/project/voxelgenerator/?et_fb=1&PageSpeed=off



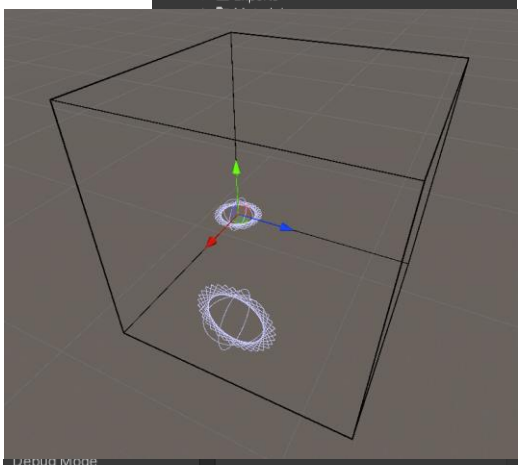


HOW TO USE:



For fast testing, just drag in one of the provided VoxelGenerator prefabs in your scene. Ideally you start with VoxelGenerator_Modular_Recommended as it contains everything needed in order to get started. Also it contains all the newest functionalities.

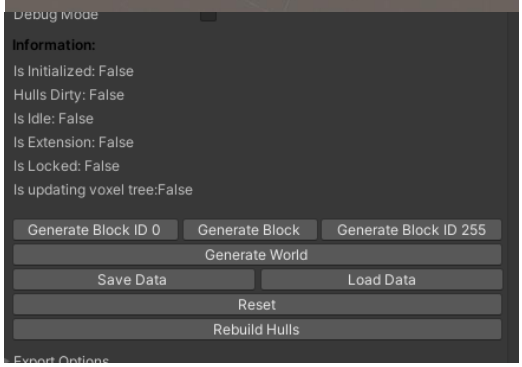
All prefabs with the exception of the "NoVisuals" version have visualizations attached so the initial block becomes visible immediately.

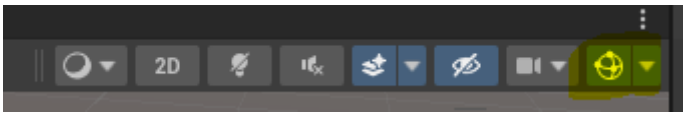


The recommended prefab contains 5 components:

1. **Voxel Generator**: This is the main voxel engine. No voxels without this!
2. **World Generator:** This system allows you to create procedural content especially terrain.
3. **Voxel Save System:** This system provides all saving/loading functionalities.
4. **VoxelModifier V2:** Main system to modify Voxels.
5. **VoxelModifier V2 Raycaster:** Allows you to paint in edit/play mode.

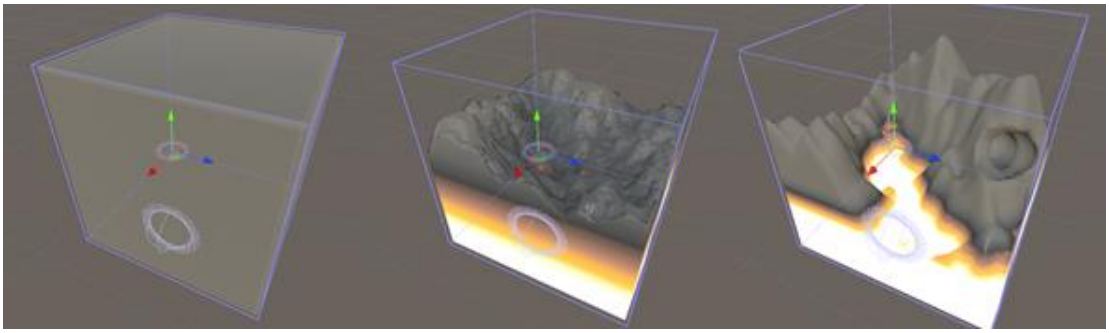
When you dragged the prefab into the scene, you will see a black frame and the Indicators from the voxel modifier. This is the uninitialized state. Enable gizmos if you don't see it.



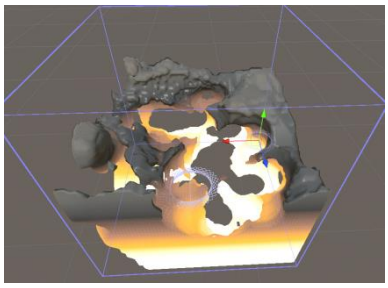


In the inspector there are buttons to initialize it. You can either load data from the save system, retrieve data from the world generator which is procedural or create an fully empty or fully filled voxel block.

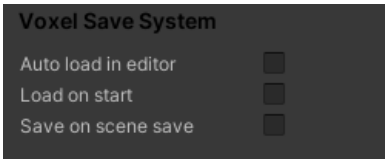
In the screen below are the results after pressing Generate Block ID 255 (left), Generate World (middle), Load Data (right)



A initialized voxel generator is visualized as light blue box. If you click “Rebuild Hulls”, you see it lighten up a bit, indicating that it is doing some work right now. Processing the Voxel data and updating the visualization afterwards.

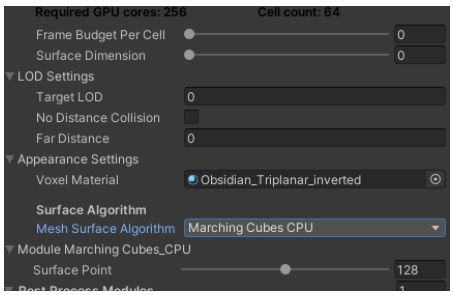


Once it is initialized, you can sculpt the volume using the attached VoxelModifier_V2_Raycaster. By holding “left CTRL, left STRG”, the light blue circle will move to the position of the cursor. With left click you are adding material, with right click you remove material and middle click smoothes the area.

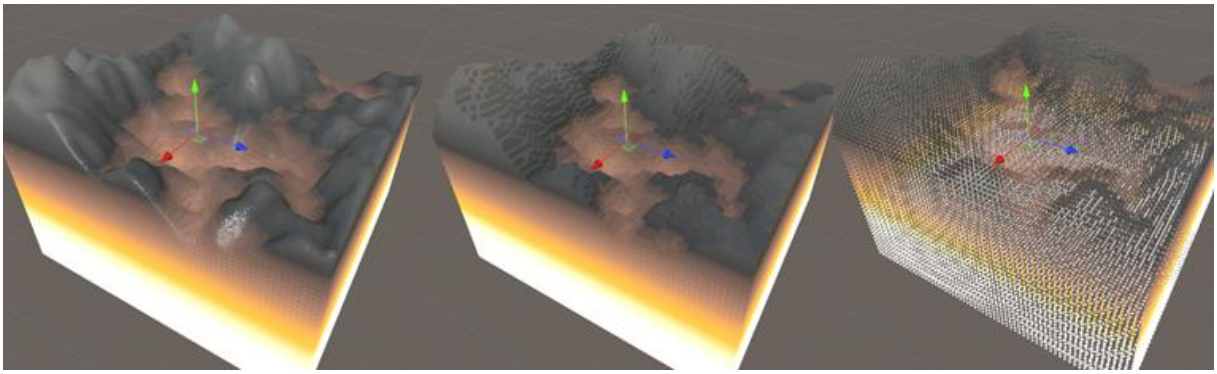


The VoxelSaveSystem provides options for saving the volumetric dataset and automatically saves the voxel data when saving the scene itself. The standard setting “Scene” is the most convenient method as it simply stores the data into the “.unity” scene file and is fine for voxel date with smaller resolution. However the Unity serialization slows down the Unity Inspector when the size of the voxel data becomes too large. For such cases, it is better to save it as ScriptableObject in order to keep the Unity scene file small. The recommended sample has disabled all Auto Save options.

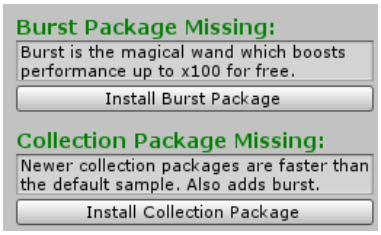
The visualization itself is separated from the main VoxelGenerator GameObject and is a separate GameObject attached as child to the VoxelGenerator. The prefab contains a child object called “HullGenerator” which is the visualization script.



When you select it, you will see that by default Marching Cubes CPU is selected which is the fastest and most optimized. You select different surface algorithms like in the image below:



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 VoxelGenerator component. Burst will boost the performance and Collections updates the internal native data structure.



MAIN COMPONENTS:

The main components are the VoxelGenerator, VoxelSaveSystem and the World Generator.

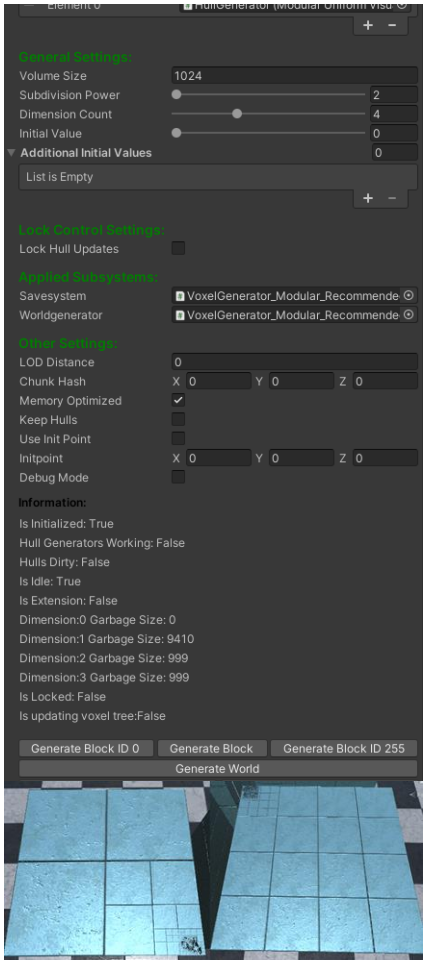
Each VoxelGenerator could exist without the VoxelSaveSystem or World Generator if saving and procedural terrain generation is not required.

Scripts which modify voxels can be attached to the VoxelGenerator but ideally are separate objects:

1. **VoxelModifier_V2:** Main component used to modify the block during edit and play mode. Don't forget having also the VoxelModifier_V2_Raycaster attached for editor sculpting!
2. **VoxelModifier(old):** Is the old VoxelModifier which is included will be **deprecated in Voxelica V2**
3. **Procedural Voxel Modifier:** Variety of modifiers or converters which are not suited for runtime voxel modification due to their complexity and computation expensive processes. Mesh To Voxel is such example!

VOXEL GENERATOR

The VoxelGenerator is the main component of the whole voxel engine and provides the hierarchical data structure for the voxel data.



Volume size: Defines the physical dimension of the voxel block, the amount of space it will occupy. The blue frame in editor view gives an indication about the size. Negative Values will not be accepted. Values between 1 and 100 are ideal performance wise. Values too low or too high could lead to floating point precision errors.

Subdivision Power: Defines the layout of the underlying data structure. The blocks in the images show the subdivision power of 2 and 4.

The most basic subdivision power of 2 represents the traditional Octree which is used in many other voxel engines and therefore is the recommended setting since it has the lowest memory footprint

Subdivision power higher than 2 changes the tree so it has 64 children instead so reads/writes are faster which is desired under advanced circumstances.

Initial Value: The initial value of a freshly generated voxel block. Value of 255 represents fully solid and 0 means usually empty. However the real behavior is dependent on the attached hull generators.

Dimension Count: The Voxel Generator has a fourth dimension. If this value is greater than one, the Voxel Generator will manage more than one Voxel Data simultaneous. The first dimension is always used to describe which regions are solid. Additional dimensions are hidden unless the hull generator uses the additional dimension for some shenanigans such as Multi Texture support where the second dimension describes the material type.

The recommended sample uses Dimension 0 for Solid Material and Dimension 1 for texturing.

Lock Hull Updates: If true, changes to the dataset will not update hull generators. The save system influences this value when loading voxel data. Voxel modification is still handled. World generators also lock the generator during generation.

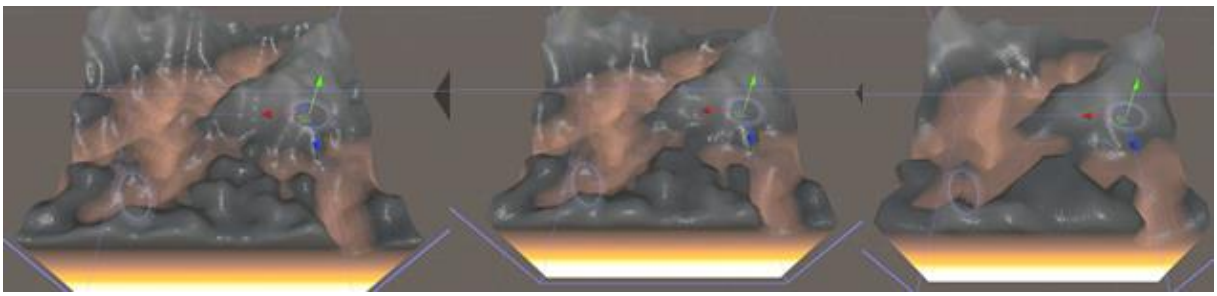
SUB SYSTEMS

Save System: Confirmation that a save system is connected to the Voxel Generator.

World Generator: Confirmation that a world generator is connected to the Voxel Generator.

OTHER SETTINGS

LOD Distance: Parameter to define LOD and can be used by Hull Generators to implement LOD. Images show the recommended sample with LOD 0, LOD 1, LOD 2.



ChunkHash: Chunk information which is important for World Generators and Multiblock Systems. You can change this manually but it may be controlled by external sources.

Memory Optimized: Forces the generator to delete lists when not needed. Highly improves memory demand but slightly lowers performance as the allocation of memory is not free. The optimization is really huge so it is recommended to have this set.

Debug Mode: Used for development and visualizes information in order to verify correctness. If true, a yellow box highlights dirty regions

INFORMATION

- **Is Initialized:** If true, generator is initialized.
- **Hull generators working:** Shows if hull generator are updating the visualisation.
- **Is Idle:** Generator is idle if no hull generator is working and also no voxel modifications have to be processed.
- **Is Extension:** True if it is an extension. Is used with infinity world systems or other third party scripts.
- **Dimension Garbage:** Shows how many voxels are in the reservoir/garbage dump for reusing. Under the hood, a reservoir provides each voxel block with voxels and the garbage indicates how many voxels are "outside" the voxel block.
- **Is Locked:** Shows if the generator is locked. The reasons are processes in the background requiring the generator to be locked until it is finished. Can only be set per scripting.

BUTTONS

Generate Block: Generates a fresh voxel block using the initial value as starting parameter Button variations with ID 0 and ID 255 exist for convenience.

Save Data/Load Data: Saves/Loads the voxel data using the VoxelSaveSystem. Is only visible if the engine has one assigned as subsystem.

Generate World: Generates a procedural world using the world generator.

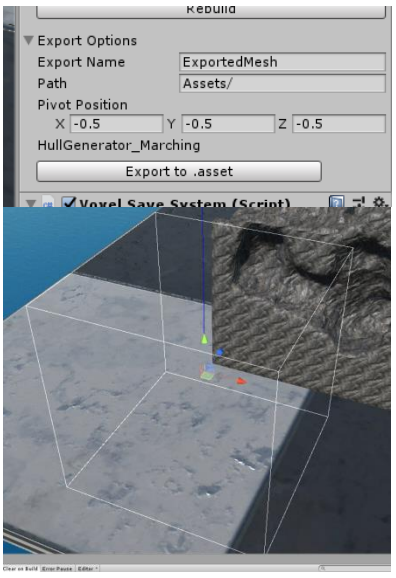
Resets: Destroys the existing voxel block.

Rebuild Hulls: Regenerate the hull generators.

Kill D(X): Destroys the specific dimension. (all voxels become 0)

EXPORT OPTIONS

The Voxel Generator also provides an option to export mesh generated by hull generators. When opening the export options, a white box appears at the center of the local position. You can define the name, path and the pivot point. By default the pivot point is (-0.5,-0.5,-0.5) which is in the center of the full voxel block. In the right image, (0,0,0) is the bottom corner and (1,1,1) is the upper right corner.



The white box is a helping indicator to see where the content will be relative to the pivot point of the game object. The right image shows that the pivot of the export product matches the center of the game object.

The exported file is a .asset file as exporting as .OBJ or .FBX is only possible with dedicated third party tools due to legal reasons.

TECHNICAL PART:

GenerateBlock(); Generates a fresh voxel block using the initialValue as starting parameter.

ResetData(): Resets all voxel maps on this generator. The initial id of the root will become 0.

SetVoxels()/SetvoxelsAdditive(): Provides direct functionality to modify voxels either directly or in an additive fashion. Basic parameter usually is the local position, target depth and the ID. These functions come in a variety which allows lists and NativeVoxelModificationData which usually is used by the save system or procedural block generators (future content).

Rebuild(): Regenerates the hull generators.

Cleanup(bool includeStatics): Destroys the voxel block. Also cleans static native containers if includeStatics is true and is only required inside the Unity editor to prevent memory leaks.

SetRegionsDirty: Marks region as dirty so the visuals have to be updated.

SetNeighbor(int ID, VoxelGenerator generator): Connect the voxel generator with another voxel generator. ID is the position where 0 is bottom left front and 26 is top right back.

RemoveNeighbor(int ID): Removes the connected voxel generator. Alternate RemoveAllNeighbor function removes all neighbors connected.

GetVoxelSize(int depth): Get the voxel size at target depth. Depth 0 is the volume size. The higher the depth, the smaller the voxels become.

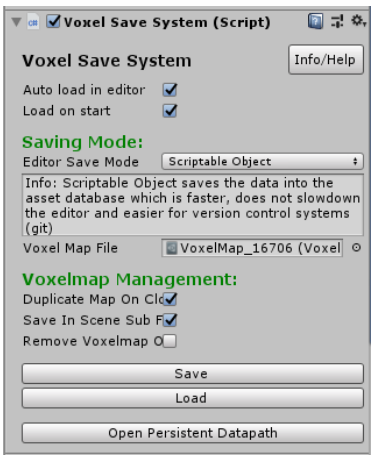
GetBlockCount(int depth): Calculates the amount of voxels the generator would have if every voxel would have the same depth.

GetBlockWidth(int depth): Amount of voxels the generator would have in one direction if every voxel has the given depth. Result * Result * Result would be the Block Count.

Show/Hide Visuals(): Shows or hide the visuals (meshes generated by attached hull generators)

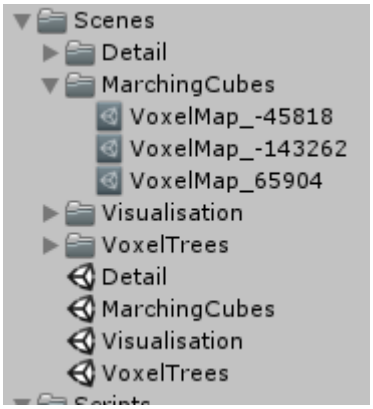
ClearMeshes(): Clears the content generated by hull generators.

VOXEL SAVE SYSTEM:



The VoxelSaveSystem is the main component for persistence and should always be together with the VoxelGenerator. 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 VoxelGenerator was modified in during edit mode, it will be marked dirty and saves the voxel data automatically when saving the scene. (pressing ctrl+s)

There are 3 different saving options where the first one is "Scene" which saves the voxel 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 voxel data is used.



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 voxel 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 voxel maps 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 voxel map if you duplicate the VoxelGenerator inside the editor. It clones the original voxel map and automatically assigns it to the cloned object. The naming is "VoxelMap" + 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 voxel 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 voxel map on delete:** If true, voxel 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 ".VOXEL" ending). It directly stores the voxel map as .VOXEL file into the persistent data path of your target device. This mode is supposed for persistence during gameplay where the player can save voxel 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.

Saving as Persistent Datapath has 2 sub variations which is **"World" and "Region".** The difference is just the naming and management in order to be able to match it to chunk positions.

"World" and is similar to "Persistence Datapath". However this saving method is supposed to be used with the infinite world system. The main parameter is the World File Name and describes the name of the infinite world. Also the voxel map file is stored as .voxelworld.

"Region" is similar to "World" and can only be used in combination with the infinite world system. The settings are identically to the "World" mode but with the addition of a World Region Size parameter. The file type of this mode is .voxelregion and each file contains the voxel data of World Region Size * World Region Size * World Region Size voxel blocks. If .voxelregion files exist at the target path, the World Region Size parameter cannot be modified.

For example if the World Region Size is 8, each file will cover 512 chunks. Therefore an infinite world has much less files which reduce hard drive read/write operations by a significant amount. This reduces performance impact at the cost of higher working memory demand since a large amount of voxel data will be stored inside the RAM.

The World Chunk is set automatically and describes which chunk is stored.

The World Folder is used when storing world information in the asset database and is assigned automatically to the scene asset location when "World To Scene Location" is flagged.

World Loading Rule describes which data storage has priority. If the infinite world is modified during edit mode, it is stored in the asset database since the persistent datapath only allows local data. When voxels are modified during gameplay, the modified data is in the persistent datapath since it was created during gameplay. The rule describes what should happen when a chunk has data in the Asset Database and in persistent datapath at the same time.

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

If **Load On Start** is true, the voxel map will be loaded on scene initialization during play mode. Also the hull generators are forced to finish their work immediately (else RigidBodies could fall through half-finished voxel blocks).

V2 UPDATE:

V2 introduces a set of new saving methods. Persistent Datapath V2, World V2, Region V2 are identical to the V1 saving modes but have significant higher performance!

New is **Save To Path V2** which simply allows you to save a file to a location defined by yourself. The editor inspector allows you to select a voxel file using the file browse.

The last addition is **Byte Buffer V2** exists for networking as it will store the data in a dictionary as byte buffer. The dictionary is static and the content is not persistent. The resulting byte[] buffer is the most granular data structure and ideal for sending to clients using your network solution. The **OnDataBufferSaved** event is fired when saving is complete and ready to be sent to anywhere.

TECHNICAL PART:

Load(): Loads the voxel data and rebuilds the VoxelGenerator

Save(): Saves the voxel data instantly. (causes performance spike)

SaveBinary(): Saves the voxel map as binary file. VoxelName member variable is used as identifier for example “VoxelMap1” without “.VOXEL” as ending.

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

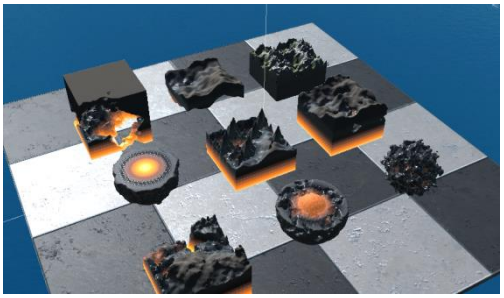
DynamicSave(): Real time saving option which does not cause performance spikes. Saves the voxel map as binary file into the persistent data path and should be called without interrupting gameplay. VoxelGenerators are Locked during saving process.

DynamicLoad(): Real time method to load any voxel map from any source without interrupting gameplay. VoxelGenerator can be modified during load but hull generators are suppressed until loading is finished.

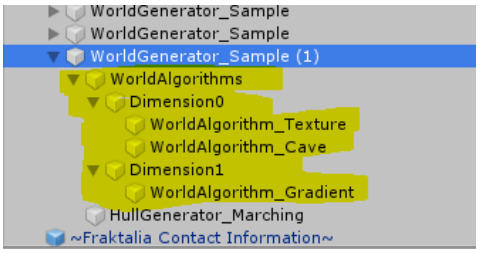
ConvertRaw(VoxelGenerator generator): Converts the voxel data of a initialized VoxelGenerator into a format which can be stored. Shrinks the data by gathering the leaves only. Returns RawVoxelData which can be saved or applied to an existing VoxelGenerator.

ApplyRawVoxelData: Applies RawVoxelData to a target VoxelGenerator

WORLD GENERATOR



The world generator was previously related to the deprecated infinite world system and became now the second sub system similar to the save system. The main difference between the World Generator and the Save System is that the Data from the World Generator is generated procedurally. The world generator will generate voxel data based on algorithms whenever the chunk which should be loaded has no saved data.



How the world is generated is defined inside a World Algorithms hierarchy. Since it is possible to use multiple voxel dimensions, the generation is separated into **WorldAlgorithmCluster** sub objects (Dimension0, Dimension1)

The **WorldAlgorithmCluster** only defines the target depth and target dimension. **WorldAlgorithms** can be attached to the cluster in order to mix different algorithms together.

Biome support probably may be implemented as derivate of a WorldAlgorithmCluster as smooth transition between 2 biomes is also required.

The world generator is applied automatically to the Voxel Generator component attached on the game object. Clicking on **Generate World** will initialize the generator and apply the world data. It also uses the **ChunkHash** parameter provided by the Voxel Generator to define the position. This is important for multi-block systems.

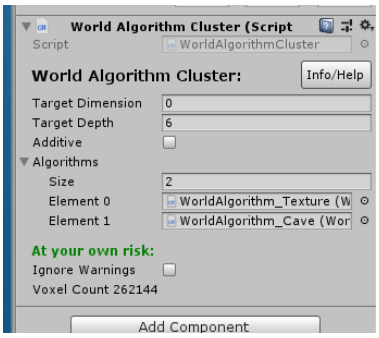
You can also apply the World Generator to any Voxel Generator using these functions:

Initialize (VoxelGenerator generator): Initializes the world generator and all algorithms attached to it. Should be called once or when algorithm parameters changed. The input generator is then used as reference.

Generate (VoxelGenerator targetGenerator): Requests a world generation for the target generator when safety is valid. Requests are added to a queue and processed one by one.

CleanUp(): To clean everything up.

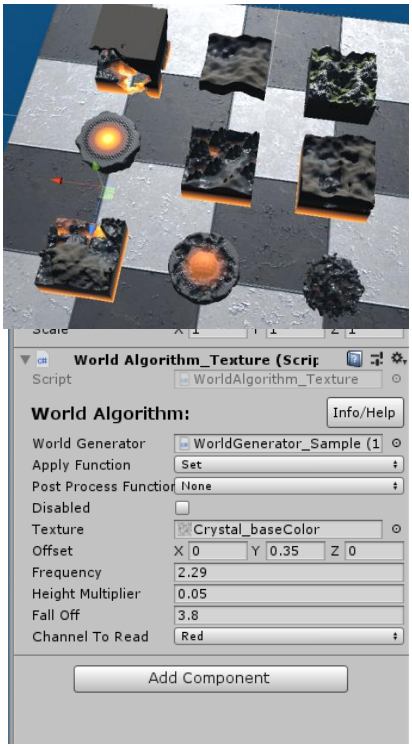
WORLD ALGORITHM CLUSTER & WORLD ALGORITHMS



The World Algorithm Cluster is the main container of World Algorithms and therefore can contain a mix of different World Algorithms. The Cluster itself only defines the Target Dimension and Depth. Modification parameters are then provided by the specific algorithms.

The sample in the right image has 2 algorithms attached as children. The first one creates a terrain based on a texture and the second one subtracts material to create caves.

Another important feature is the auto updating system which helps designing the procedural. It has a very strict safety system as it also updates the result when changing the Dimension and Depth. Increasing the Depth otherwise could quickly freeze Unity3D.



The world algorithms are the main components for terrain generation. There is a variety to choose from and is regularly expanded.

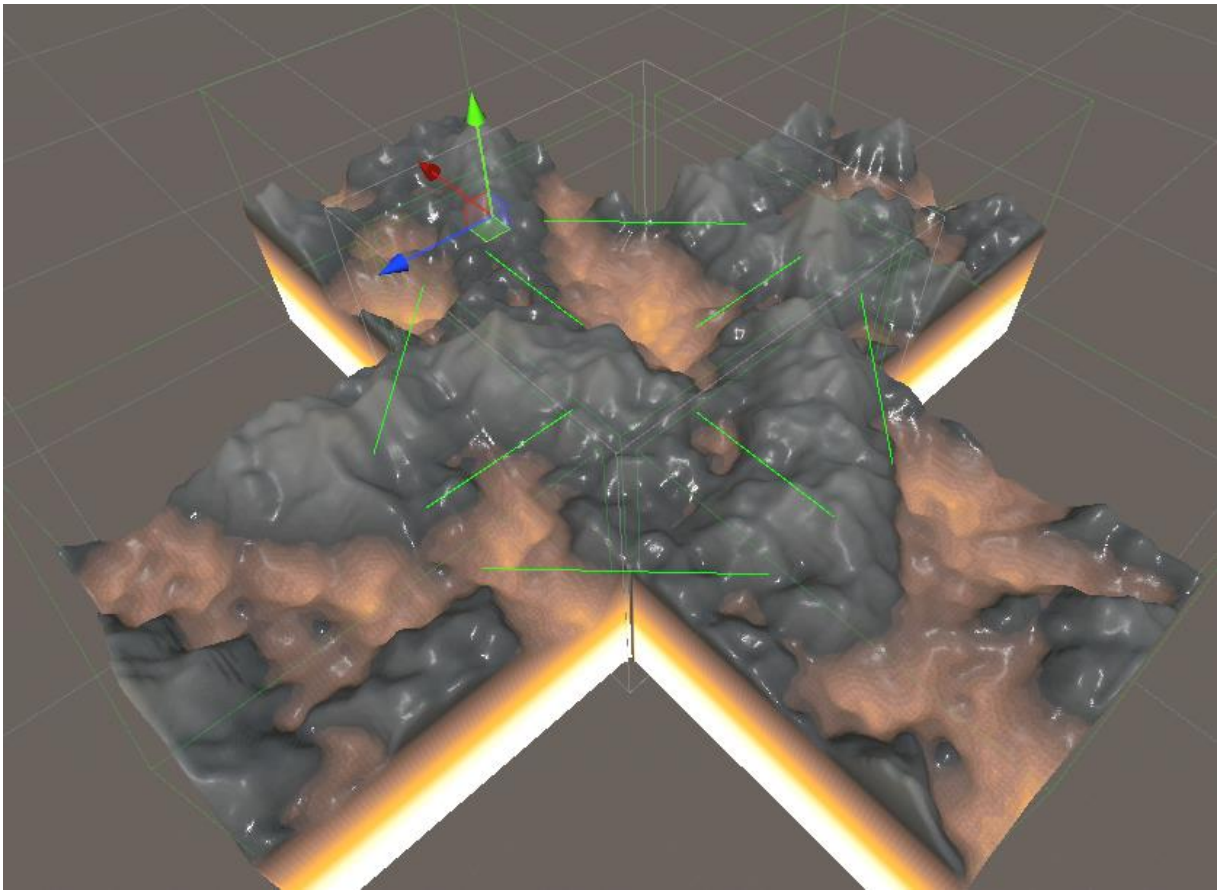
Describing every algorithm implemented is pretty overwhelming so it is easier to check the sample scene and experiment with them as they are updating automatically.

Every algorithm has an Apply Function which is important for mixing.

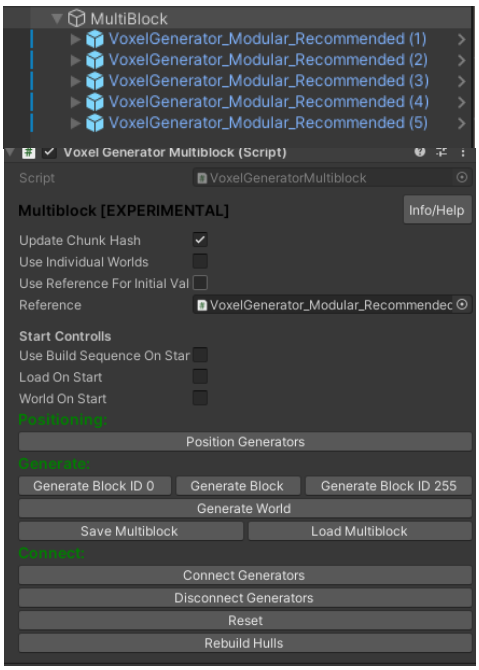
The current options are Set, Add, Subtract, Min, Max and their inverted versions. The first algorithm usually has Set. The inverted versions apply 255 – value instead.

It is also important that some Apply functions are highly sensitive to negative values especially Add/Subtract. Therefore a post process function is also included.

MULTI BLOCK



The multi block is one of the recent addition to Voxelica and allows you to connect multiple generators together. I In order to utilize this system, create a game object and



add the VoxelGeneratorMultiBlock script.

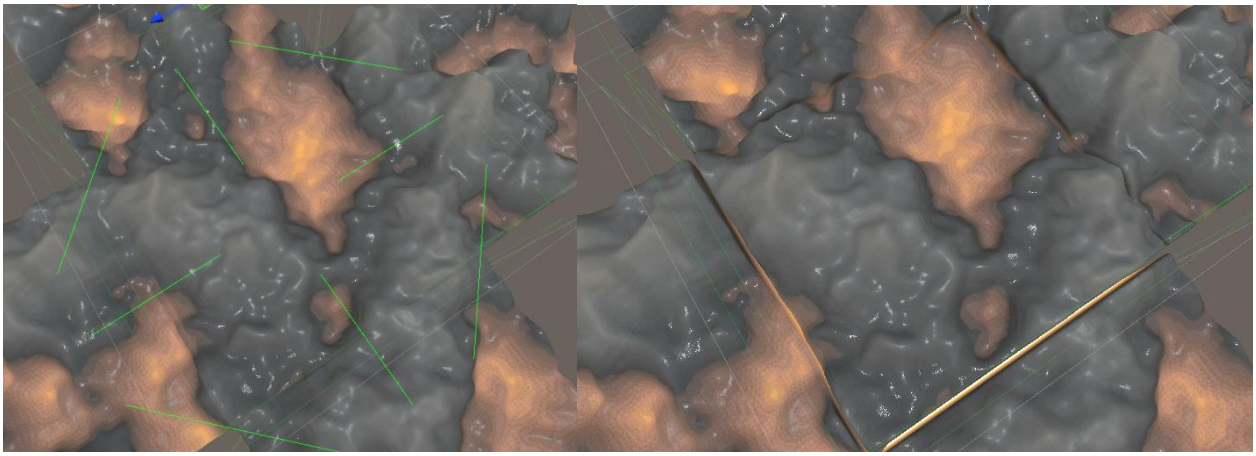
Afterwards attach one VoxelGenerator to it as child. Then assign this the attached VoxelGenerator as “Reference” to the VoxelGeneratorMultiBlock script.

The VoxelGenerator assigned in the Reference field acts as reference and marks the center chunk of the multi block setup. After the assignment, a subtle white and yellow box frame becomes visible. The reference has a white box additionally to the yellow one

Now duplicate the reference and move it away from the center. It will try to snap into a chunk position. You can create as many as you want and also move them in the Z axis.

Afterwards click one of the Generate Buttons. It will initialize all generators. After initialisation, you will notice that there are gaps between the hulls. This is because they are now initialized but not “Connected”. In order to connect them, click on “Connect Generators”.

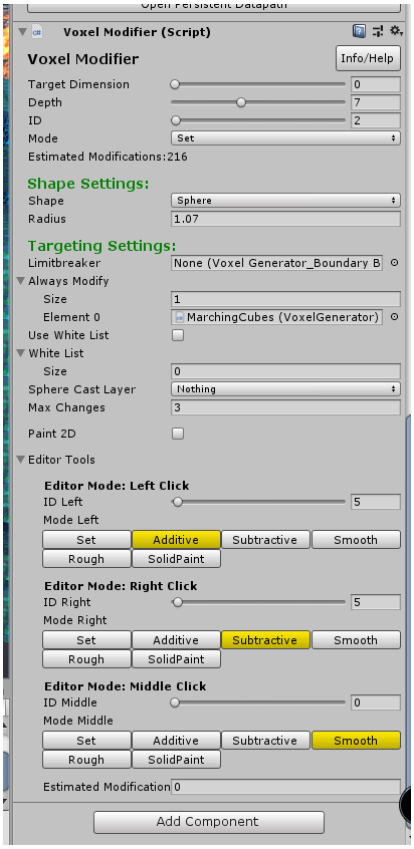
When they get connected, you will see the gaps disappearing as the hulls are updated! Also the lines between the generators show that they are connected.



INFORMATION

- **Update Chunk Hash:** If true, the Chunk Hash is updated according to the position relative to the reference. If you are using world generator, it makes sense to have this checked otherwise the world will repeat itself.
- **Use Individual Worlds:** Another setting for the world generator. If unchecked, the world generator of the reference is used. Otherwise individual world generators are used instead.
- **Use Reference for Initial Values:** Generators are initialized with their initial value being usually 0. Having this checked overrides the initial value of all chunks.
- **Use Build Sequence On Start:** If checked, it will initialize on start.
- **Load On Start:** Apply save system on start.
- **World On Start:** Apply world generator on start.

VOXEL MODIFIER

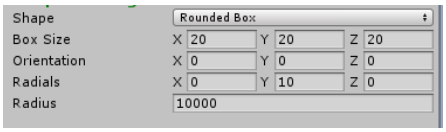


The VoxelModifier is the third component and the main tool to modify voxels. This script allows painting during edit mode and provides other functionality to modify the dataset. It allows 3D painting on solid geometry but also supports 2D painting in cases where you want to use Voxel for 2D games. The most important parameters are the radius and depth. A safety system is implemented which prevents insane settings like radius 10000 and depth of 20.

Target Dimension: The dimension which should be modified. Standard value is dimension 0 which is the normal Solid-Nonsolid voxel map. What exactly is modified is highly dependent on the hull generator attached. Dimensions greater than 0 may be used for material, temperature, humidity, holy/unholy ground etc.

Depth: The target resolution.
ID: Independent ID used to modify the dataset when ModifyAtPos is executed.

Shape Settings:
The main basic shapes are Sphere, Box and Rounded Box and Single:



Sphere is the most basic form and is only defined by the radius parameter.
Box is defined by the Box Size which defines width, height and depth and a rotation parameter.
Rounded Box is similar to Box shape but has additional parameters for radials. This option provides the maximum amount of freedom without using more complex functionality. The sample image shows the settings for a cylinder with a radius of 10 in Y axis.

Single Modifies one single block (Minecraft Style). This mode has no parameter as it only modifies one single voxel according to the given world position.

The function called WorldPositionToVoxelPoint is used to evaluate the exact position of a voxel according to the target depth and returns the voxel position as world coordinate. Additionally a **normal direction** and **normal offset** can be used to apply an offset.

In order to simulate Minecraft like behavior (left click destroy, right click place), the **normal offset** should have a value of -0.5 when destroying a block and 0.5 when the user places a block with right click. The **normal direction** must be fetched from the raycast hit information.

Estimated Modification Count: Indicator of the safety system which shows how many voxels would be modified. Safety limit are 100000 voxels per modification. The referenced Voxel Generator is the one which is targeted first as multi block editing is possible.

Modes: There are separate modes for left and right click including a separate target ID applied.

- Set: Sets the voxel ID to the target ID.
- Additive: Adds the target ID to the voxel ID (target ID can be negative)

- Subtractive: Subtract the target ID to the voxel ID (target ID can be negative.
- Smooth: Applies mean filter to the target region. Target ID is used as strength.
- Rough: Unsmooth target region. If Voxel ID is greater than Target ID, it becomes 255 else it becomes 0.
- SolidPaint: Changes voxel ID to target ID if voxel ID is not zero (for AtlasCubes painting)

This component can be used to modify multiple VoxelGenerator at once. Therefore the VoxelModifier has a variety of options to determine the targeted generator. The simplest way to assign a generator is by simply adding one or more VoxelGenerator to the "Always Modify" list.

If **White List** is true, only VoxelGenerators assigned to the whitelist will be modified if they are fetched by the sphere cast.

The **Sphere Cast Layer** defines which layers can be hit by the sphere cast.

MaxChanges limits the amount of voxel generators which can be modified at the same time.

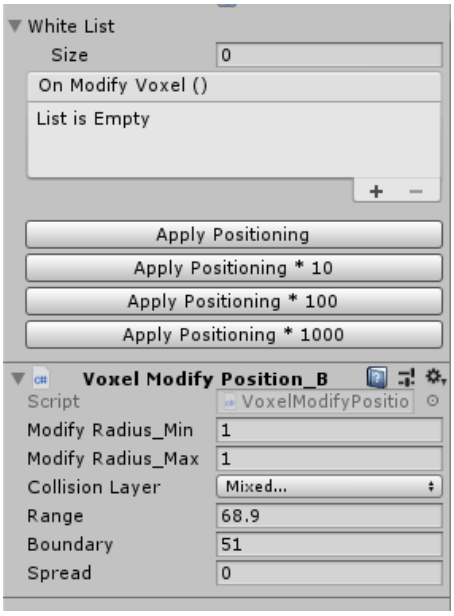
LimitBreaker: Is required when infinite world mode is active. When a VoxelGenerator_BoundaryBreaker script is assigned, the automatic fetching system is determined by the infinite world. This allows boundless sculpting.

Editor Tools: These are entries for editor sculpting and provide an option for left, right and middle mouse clicks. Each option has an ID and Mode parameter.

The main function **"ModifyAtPos"** will execute a modification command and requires the target world position as input parameter. This function is called by the sample controller whenever the player holds CTRL while clicking on a solid surface. This function can be called from anywhere such as a projectile calling this function on impact to destroy the environment. Usually you can also set ID and Radius to the desired values before calling it.

Additionally an "ApplyPositioning" function is included which requires a VoxelModifyPosition component and can be used for procedural voxel map generation.

POSITIONING ALGORITHMS



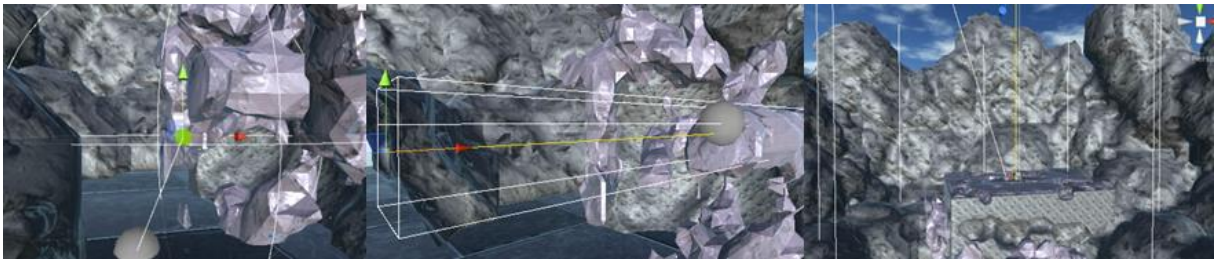
The **VoxelModifyPosition** are components used for procedural positioning for modifications. The sample scene Marching Cubes contains a snow producer GameObject which uses such component.

If the GameObject containing a VoxelModifier component also contains a VoxelModifyPosition, the VoxelModifier component will contain buttons to apply the positioning algorithm during edit mode.

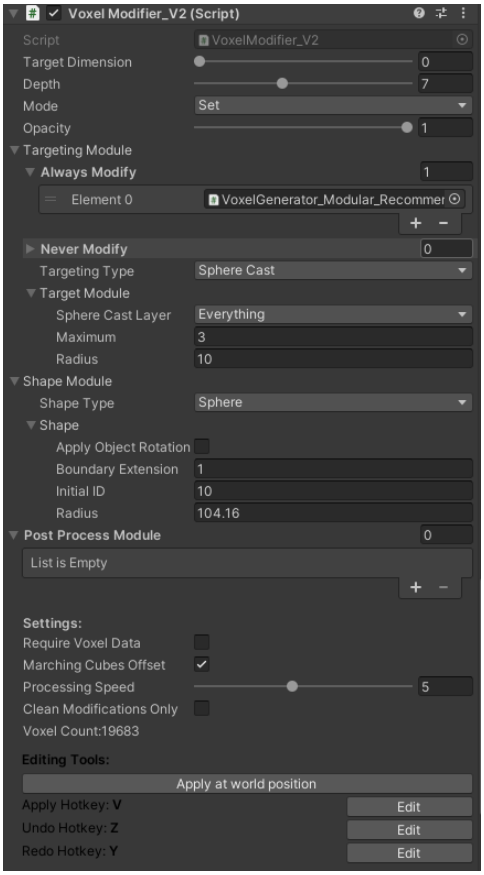
The main parameter of a VoxelModifyPosition is the ModifyRadius Min/Max for randomness and the collision layer. When such GameObject is selected during edit mode, a preview will indicate where the positioning algorithm could be applied.

Currently included positioning algorithms are Sphere, Beam and Box. The spherical positioning can be used for explosion effects as It does ray casts from a spherical direction. Beam and Box are “lasers” where ray casts are shot in one direction where Beam is cylindrical and Box is rectangular.

Voxel Modifier V2 also supports this.



VOXEL MODIFIER V2



The VoxelModifier V2 is a complete overhaul of the old VoxelModifier. It has Undo/Redo functionality and allows the alteration of modifications such as Hardness, Filling, Destruction and is constantly expanded with new content. It also has a significant improved performance.

Overall it is a modular system comprising of 3 modules which are: Targeting Module, Shape Module, Post Process Module.

The **Target Module** determines which Voxel Generators can be modified. Usually you have those assigned in the "Always Modify" script but fetching them using sphere cast is also possible.

Shape Module: This module determines the shape. Sphere, Box and Ellipse are the most generic ones. Future updates will introduce additional shapes.

Post Process Modules: Post process modules alter the modification itself. There can be more than one post process module. However some modules like "CopyPaste" will interfere with previous post processes.

The samples in the **VoxelModifier V2 Use Cases** show the effects of the individual post processes.

The primary settings like Target Dimension, Depth and Mode are identical to the old VoxelModifier. Additionally it has some very important new settings:

- **Require Voxel Data:** Some post processes are dependent on the current voxel data in the affected area before it is modified. For example ShapeFill has to check the values of current voxels in a "Shape Dimension" in order to determine if voxels should be modified or not.
- **Processing Speed:** Multiple modification calls are queued. Value defines how many modifications are processed per frame.
- **Clean Modifications Only:** This is important if post processes have rules which could be breached by using outdated voxel data while the engine is updating the data. If true, the modifications will be foolproof if the current data of the dimension which is modified at the same time is important as they are only processed when the data is not dirty. However, the data processing will lag behind.

One important difference is that it can modify the position at its own location which is important for stuff like "Bombs" which should modify the data at the location of the bomb when it explodes. In editor, you can click on apply at world position or by pressing the Apply Hotkey.

However by default this component alone does not implement editor painting! In order to enable editor painting, attach the VoxelModifier_V2_Raycaster.

VOXEL MODIFIER V2 RAYCASTER

This script implements editor sculpting like with the normal voxel modifier. It is similar to the built-in editor painting tool included in the old voxel modifier.

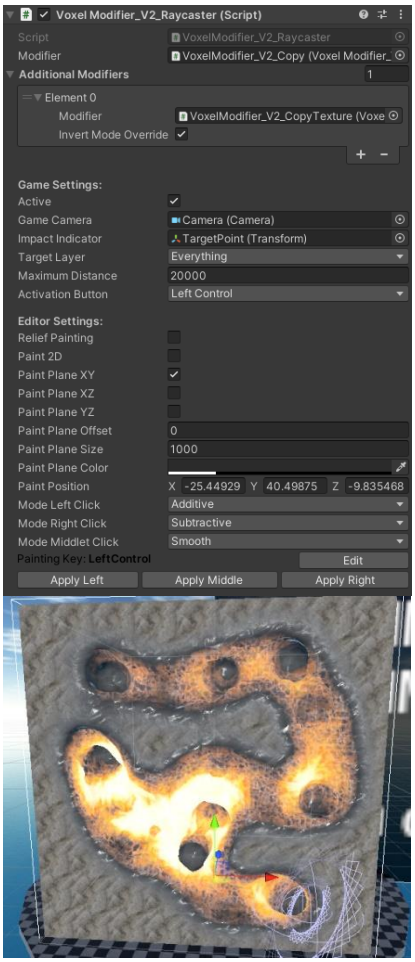
The main addition is that you can assign additional modifiers so it will apply multiple V2 Voxel Modifiers at the same time. The example in the right image has 2 modifiers assigned. The main modifier modifies dimension 0.

The additional modifier modifies dimension 1. Additionally the second modifier has **invert mode override** checked. This check inverts the additive/subtractive mode so in this case, when material is added, the values in dimension 1 are subtracted.

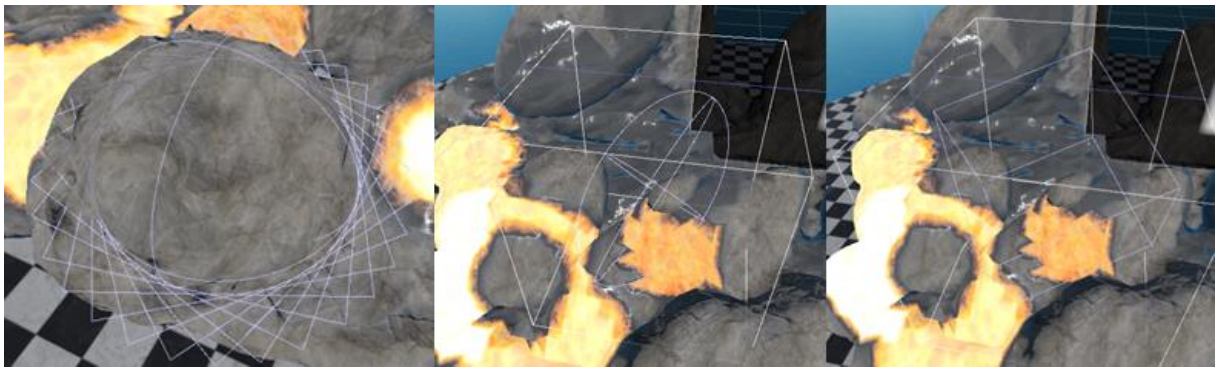
The right image shows the result when applied. It “Melts” the surface becoming lava like while material is removed. Surface is solidified when material is added.

Besides editor painting, it also has settings for usage ingame if active. Since you don’t have gizmos during play mode, you need a **target point** to indicate where the modification will be applied. Also you can define a target layer and a maximum distance.

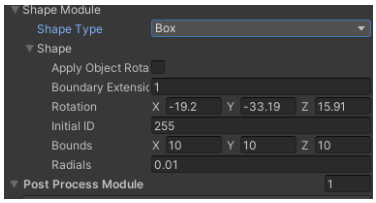
The game camera is required for the raycasts (mouse to object surface).



SHAPE MODULES



The shape module defines the shape of the modification. The most generic ones are Round, Box and Ellipse. Every modifier has exactly one assigned. When selected, the light blue area indicates where voxels are modified. The white box indicates the area voxels are at least “touched” but may not be modified as they are outside the light blue area.

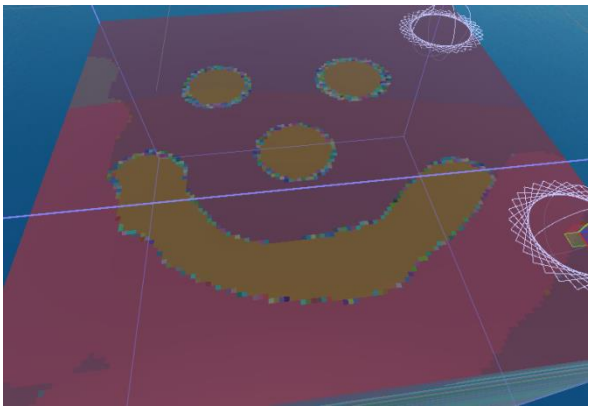


All shapes except sphere have the rotation property and the white box is adapted automatically so the rotated area fits in. Boundary extension of 1 is recommended in order to ensure that the modification is not cutoff

Initial ID is like the opacity value. A lower initial ID means that the modification is much softer with smooth removal/addition of material while a huge initial ID makes it very rough and sharp. The shapes itself are pretty straight forward.

Future updates may introduce other shapes as with V2, more complex shapes are possible.

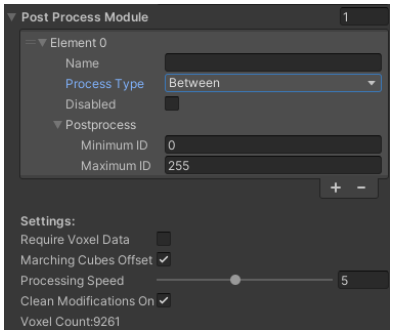
POST PROCESSING MODULES



A modifier can have post processing module applied in order to further modify the way material is manipulated. One of the most basic example is the solid paint which has uses the “SolidPaint” post processing module.

This module is useful when the visualization is cubic where 0 is empty and 1-255 is solid while the value is used as lookup index in a texture atlas. (Minecraft)

This post process now forbids the modifier from modifying air (prevent voxels from being modified if their current value is 0) which now allows to paint the surface with whatever ID value desired.



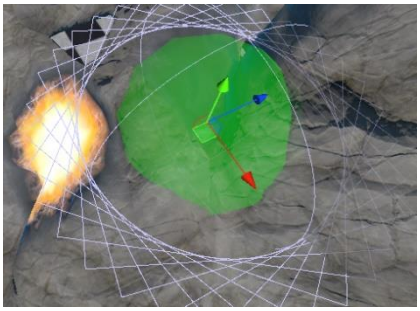
Also whenever you are applying post processing modules which depend on the current voxel data, it is recommended to have “Clean Modifications On” as it forces single modifications at the same time. Otherwise you might experience race conditions.

For example rule forbids reducing ID of a voxel lower than 1 but is applied twice. Before applying, the ID of the affected voxels is 2. If both are applied simultaneous, both rules would be valid and each modifier would reduce the ID by 1. The resulting voxel now has an ID of 0 despite the rule forbidding the reduction of an ID lower than 1. With Clean Modifications On, the engine will wait until the first modification is done and then the second one is applied.

BETWEEN

Between is a simple post process which prevents a voxel to become lower than a given Minimum ID and higher than a Maximum ID. If the voxel value is outside the range, it is clamped to be in the range.

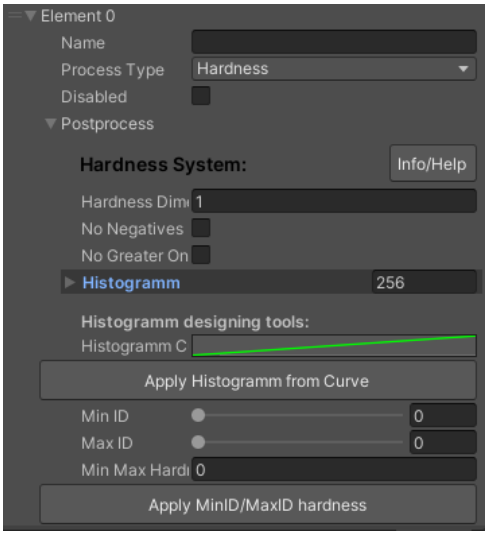
COPY PASTE



This post process implements copy and paste functionality. When applied, it requires copied data. The raycaster will recognize it and allows you to copy the surface with “CTRL-C”. After that you can paint as usual using the copied information.

The greenish sphere indicates when the modifier has copied data.

HARDNESS (ADVANCED)



This post process is quite advanced as it introduces hardness, making voxels with specific IDs resistant to modification.

Ideally the dimension defining hardness is not the surface dimension but something used for texturing. If the dimension used to evaluate hardness is the same as modified, clean modifications should be enabled.

In order to apply this post process properly, you need to provide a histogram which contains multipliers for all IDs between 0-255.

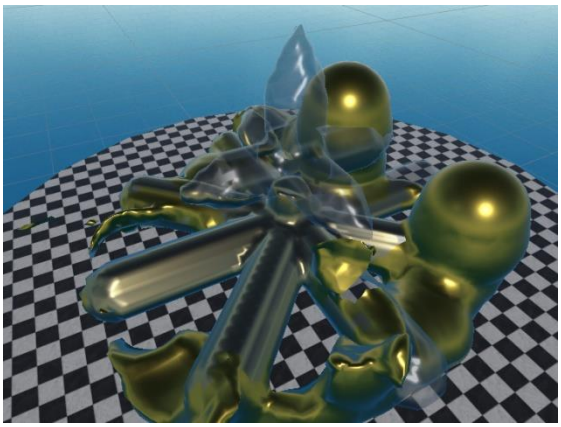
The editor provides some helpers to create such histogram from a curve. But you can also create it manually.

In the end the values in the histogram are multipliers like opacity while the index is the Voxel ID where this multiplier is applied when modified.

A value of 0 makes Voxels with the affected ID immune to modifications and 1 is default.

Since negative and values greater than one provide weird results, you have the option to prevent those.

INDESTRUCTIBLE



This post process will define a Shape dimension and a Density dimension. The Density dimension is matched against the Shape dimension and ideally matches the Dimension which should be modified.

In the end this post process, the shape dimension defines the final shape when this modifier attempted to remove all voxels in the entire volume.

The right sample has 2 hull generators. One for D0 and one for D3 to highlight the shape.

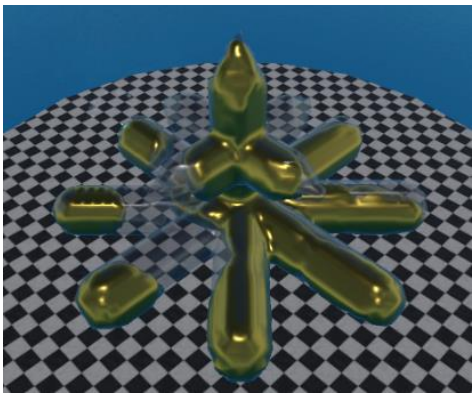
Ideal usage for this is when the user has to dig out a fossil.

SHAPE FILL

This post process is doing the opposite of “Indestructible”. It only allows the modifications of voxels inside the shape volume.

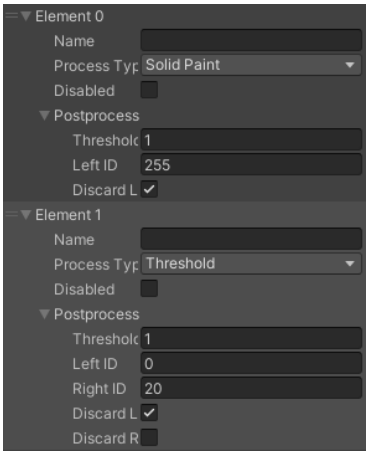
You also define the Density dimension and the Shape dimension.

The ideal usage for this is if the user has to “Craft” an object using additive methods. So the user has to add material in order to create something.

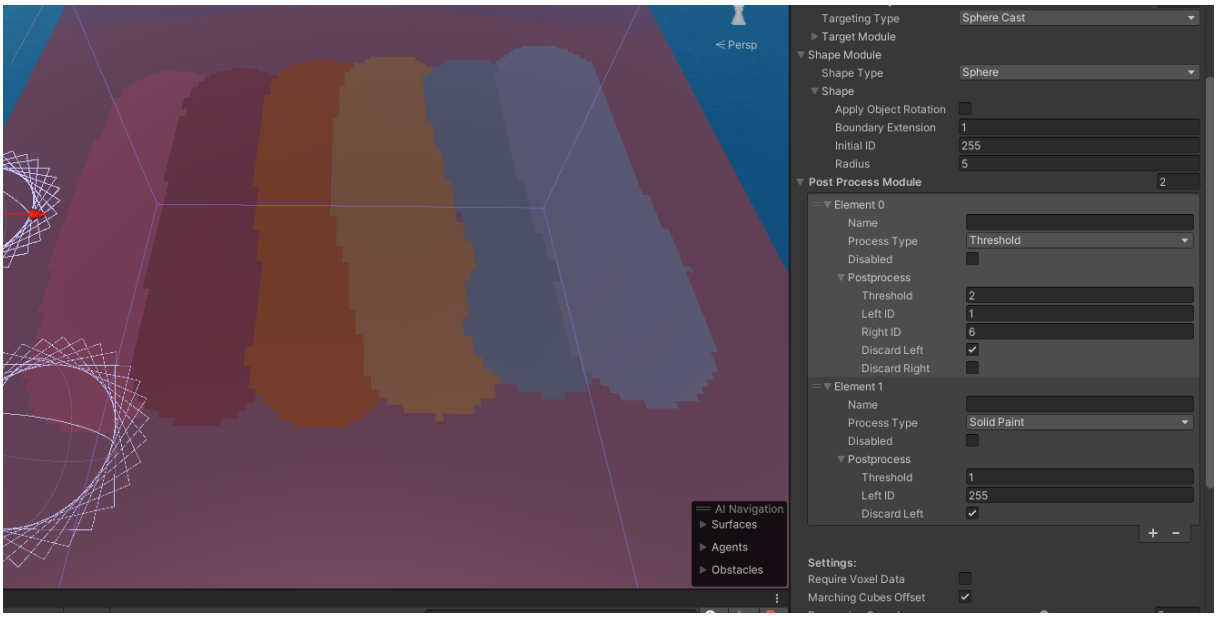


SOLID PAINT

This post process prevents voxels being modified if the resulting ID would be smaller than the Threshold. The example with threshold of 1 prevents voxels from becoming 0 as “Discard” is checked.



THRESHOLD



Threshold post process simply changes the values of the modifier data. The values lower than the threshold become the “Left ID” and those higher become “Right ID”. Usually the Shape module produces a gradient. With threshold, this gradient is removed.

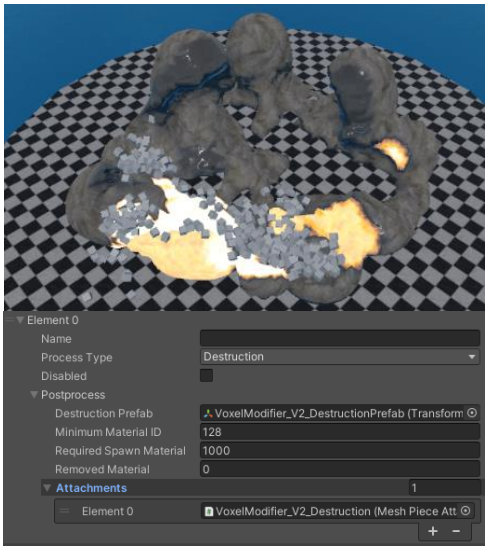
Ideally this is used with “Set” mode in combination with SolidPaint like in the image above which. First the threshold removes all values which are on the left side of the threshold as discard left is checked. All those on the right are set to “6”.

The image shows 6 lines (most left is hard to see) and have the IDs between 1-6

Solid paint then checks that the values are indeed solid.

The mode of the modifier is “Set”

DESTRUCTION (ADVANCED)



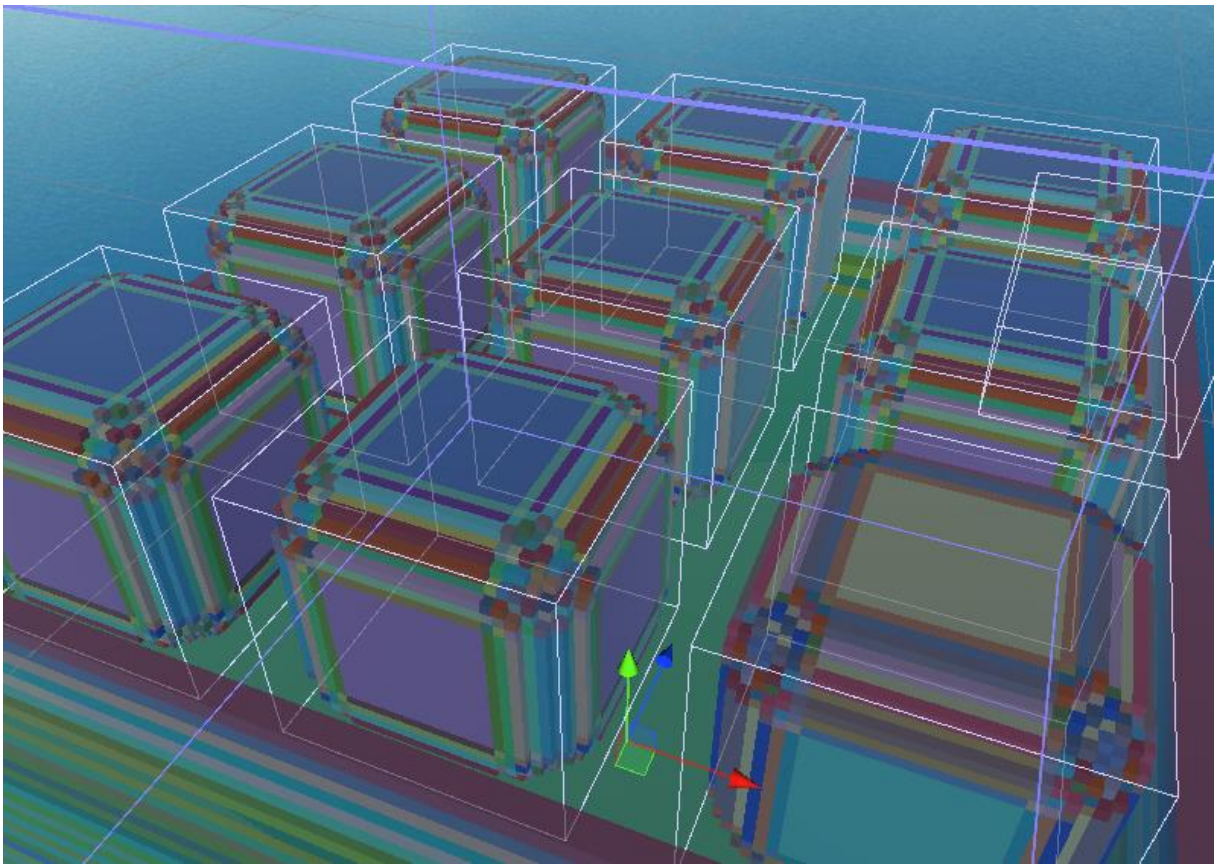
This modifier will create instances of the assigned prefab when material is removed. The volume removed is used to determine the amount of material which must be removed.

The Minimum Material ID defines that only Voxels which an ID greater than this value will count towards the total volume removed in order to prevent “Invisible” material from creating prefabs.

Required Spawn Material defines how much volume must be removed to create one instance. Be careful with low values as there is no safety system regarding this!

You need a prefab assigned which also can have MeshPieceAttachments. In this case it has a “Decay” attachment so those instance disappear after a while.

VOXEL MODIFIER V2 CALLER



This simple script allows you to call multiple VoxelModifier V2 game objects simultaneous. This is useful when you use VM2 as tool for level design.

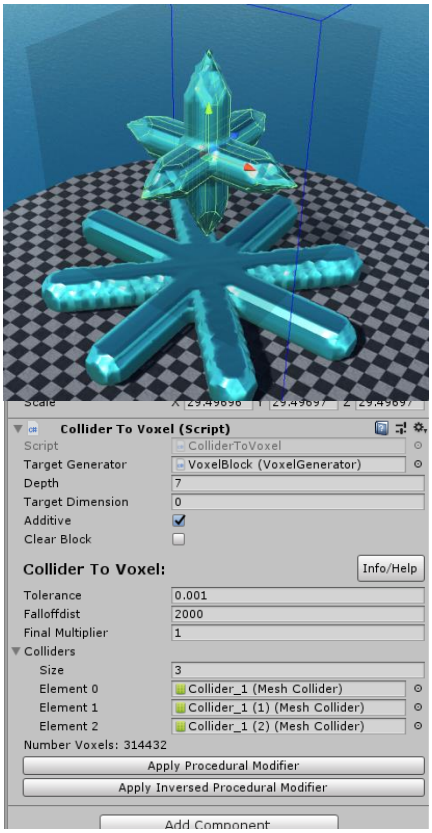
PROCEDURAL VOXEL MODIFIER

Procedural Voxel Modifier is a group of scripts supposed to modify the Voxel Data using algorithms and are completely independent.

Every Procedural Voxel Modifier has the **Target Generator, Depth and Target Dimension** as main parameters. Additionally you can decide if the result should be applied in an additive way or not and if the block should be cleared completely before applying the modifier (for biomes).

The main aspects of procedural modifiers are that they have fairly complex computations when applied or convert external data to a voxel representation somehow. Modifying the voxels itself is the insignificant compared to the process defining what and how something should be modified. Therefore these are not suited for real time usage and it is recommended to apply them during level loading.

COLLIDER TO VOXEL CONVERTER



This procedural modification script converts any Unity3D collider into a voxel representation. Any collider attached as child to the GameObject will be used when the modification is applied.

Mesh colliders must have “Convex” enabled. Therefore the best way to convert a concave model into a voxel representation is to create several convex mesh pieces and attach them to the converter.

The first parameter “**Tolerance**” defines how close a voxel must be to the collider that it becomes solid. This value usually is very small otherwise the result will become less accurate.

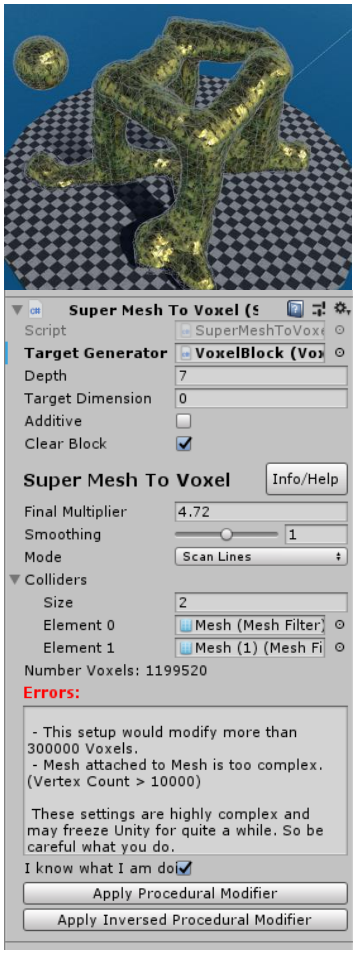
Falloffdist defines the solidity falloff when a voxel is not close enough to one or more collider. Large values cause a sharp falloff which makes the result blocky. Ideal value is between 1000 and 10000 which creates a smooth result.

Final Multiplier is a simple multiplier. When additive is set and the value is negative, material is removed instead.

In the editor window, the darkened array gives an indication about the voxel area which will be modified. The safety system limits the amount of voxels which can be modified to 5 million.

SUPER MESH TO VOXEL CONVERTER

This mesh to voxel converter was an attempt to improve the original mesh to voxel converter by implementing a custom evaluation method to determine if a voxel is inside or outside the mesh geometry. The functionality is identically to the original mesh to voxel converter but the parameters are slightly different:



Final Multiplier: Final multiplier of the values.

Smoothing; Value defines the smoothing filter applied to the result by applying an average filter over the result.

Mode: Evaluation algorithm used to define if inside or outside.

Brute force simply checks every voxel if it is inside or not. Brute force mostly is faster on simple meshes because it is very cache friendly especially when the evaluation is cheap.

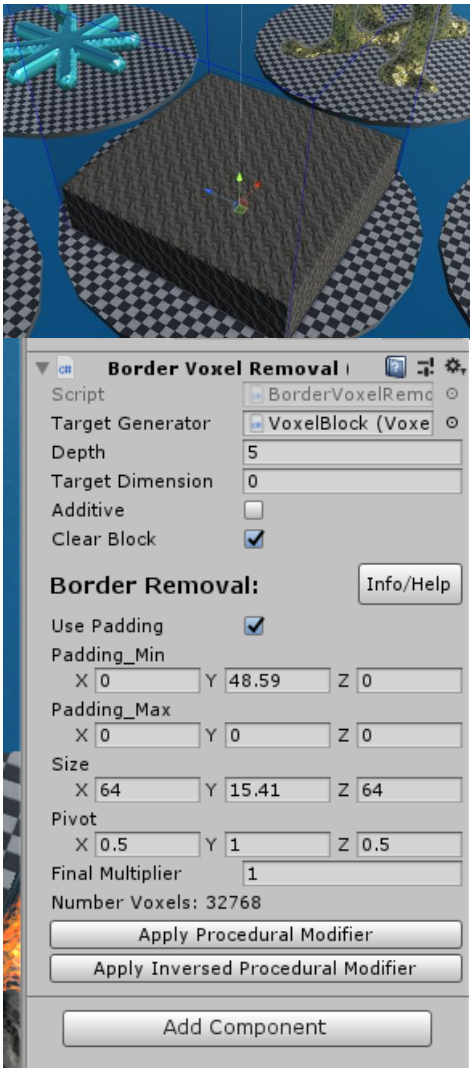
Scan Lines uses the scanlines algorithm to convert a mesh to a voxel representation. It has to build the tree first which is costly but then the evaluation is much faster than brute force. Therefore it is faster when the mesh has a higher vertex count.

It is important to increase the depth only step wise as the number of voxels increases exponentially. Also the resolution of the hull generator should match the target depth in order to fully see the result.

BORDER REMOVER

This procedural modifier does nothing more than removing the border region of a block. And is the primary method to initialize a voxel block with a defined block shape.

It only has values to define the padding for each side but also has the option to use Size and Pivot Point. The affected volume always matches the volume size of the Target Generator.



TEXTURE TO VOXEL

Texture to voxel converts a set of textures into a voxel representation. This modifier is very efficient and is able to generate highly accutate results. The result is generated layer by layer like a 3D printer because it uses the slices as input parameter. Every slice requires a texture assigned and defines one voxel layer.

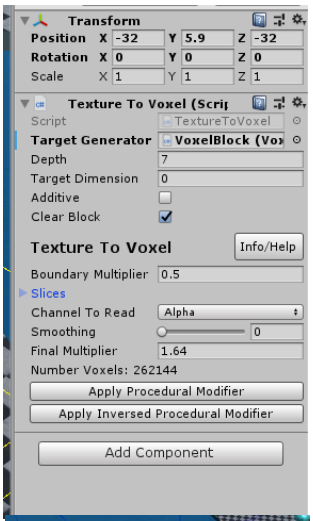
The right sample uses 32 slices in order to create the ornament like structure. In this case all 32 slices have the same texture(third image) assigned. Also it is possible to assign different textures for each slice so it is possible to visualize scan systems which create image slices.

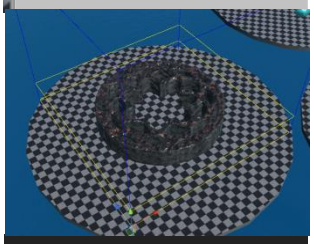
The amount of voxel is automatically calculated using the depth and boundary multiplier so it fits into the voxel block perfectly.

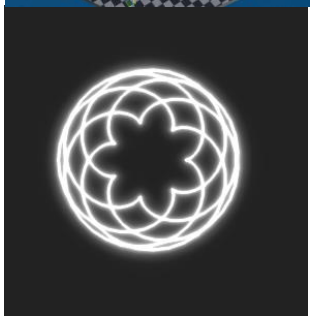
The boundary multiplier is used to oversample or undersample the measurement as creating a 3D object using 2048*2048 may be too much.

You can also select which channel you want to read. Scans are usually in gray scale where it has no influence.

Also you can smooth the result afterwards which removes noise if your slices suffer from noises.

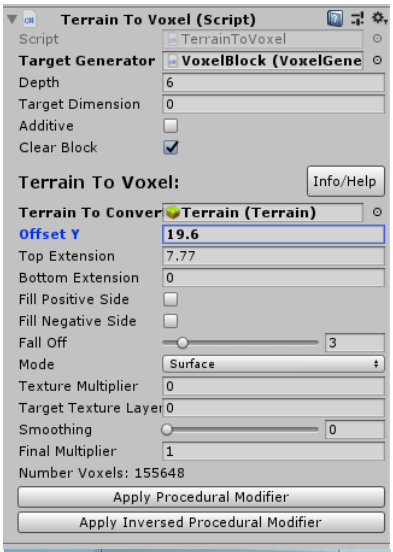




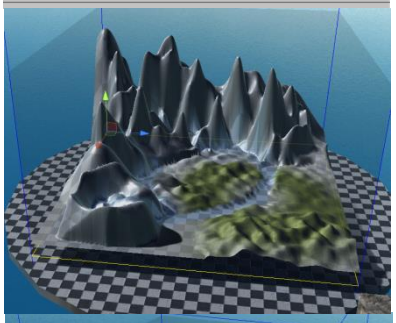


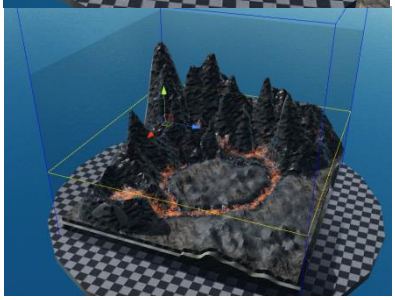
TERRAIN TO VOXEL

The terrain to voxel converter was a feature requested by customers and allows the conversion of Unity terrain into a voxel representation. One key aspect of terrains is that they have no 3D layout so it must be extrapolated with parameters. Also terrains have multi texture support which means that every texture added inside the terrain editor has a separate alpha map.



Therefore to fully convert a terrain into a voxel representation, you need 2 Terrain To Voxel modifiers. The first one has Surface mode selected and writes to Target Dimension 0 in order to create the surface geometry. The second modifier has to read the texture heightmaps of the terrain in order to write into the texture dimension (target dimension 1 or higher). The mode must also be set to “Dominant Texture” or “Individual Layer”. Also the generator, modifier, and terrain ideally have the same world position like in the right images.





This modifier has 3 modes. The first one is **“Surface”.** Surface reads the heigth map in order to create a 3D gradient which can be moved with the **Offset Y** parameter. Also the modifier automatically calculates the boundary which can be extended using **Top/Bottom Extension**

Fill Positive/Negative sides would completely make voxels below or above 0 completely solid if set.

The **Fall Off** defines how fast the Density decreases the further away the voxel is from the surface.

Mode is the most important setting as it defines the evaluation method:

- **Surface:** Reads the height value of the terrain. Used to create the solid geometry.
- **Dominant Texture:** Reads all texture layers and uses the index of the texure layer with the highest value. Used for multi texture lookup and is multiplied by the Texture Multiplier parameter.
- **Individual Layer:** Reads the value of the layer defined by Target Texture Layer

As usual, the result can be smoothed by increasing the **smoothing** parameter.

TEXTURE TO VOXEL TERRAIN

This modifier is identically to the Terrain to Voxel modifier as it converts a 2D map into a voxel representation. The main difference is that it uses a texture as input parameter instead.

It has the same paramters as the Terrain to Voxel modifier.

First you have to select the channel you want to read. Options are Red, Green, Blue and Alpha.

Invert inverts the evaluation value. (0 becomes 1, 1 becomes 0).

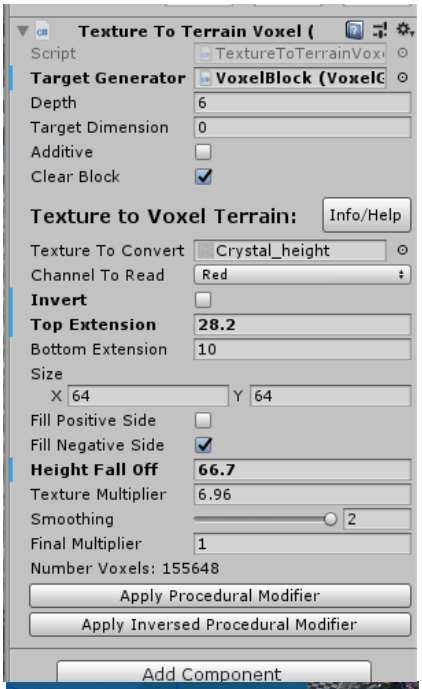
Size defines the X and Y size of the final result. The lookup is adapted automatically.

Height Fall Off defines how fast the voxel ID decreases the further it is away from the surface. If 0, the result has no fallof and behaves like **texture to voxel** modifier .

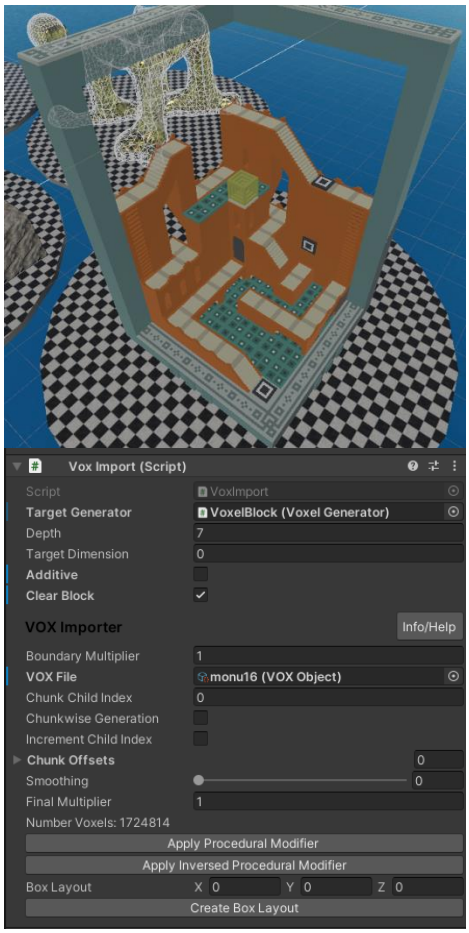
Texture Multiplier is multiplied into the texture value.

If the result should have solid ground, setting Fill Negative Side is recommended, else the result will be a pillar.

Unless Terrain To voxel, this modifier has no mode as there is only the assigned texture to read. The right image is created using 2 modifiers where one writes Target Dimension 0 and the other one Target Dimension 1.



VOX IMPORTER



The VOX Importer allows the import of .VOX files which was introduced in MagicaVoxel

Source (https://ephtracy.github.io/)

It is faily simply. Apply a .VOX file from the database and make sure that it is inside the boundary of the voxel generator. Then hit the apply button.

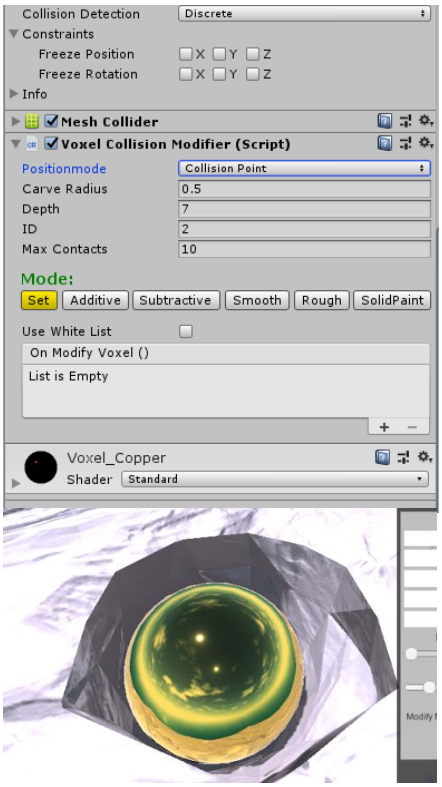
Some .VOX files have chunks. In such case you can set Chunkwise Generation to true and define the chunk offsets for each chunk. In the .VOX file, the chunks only have an index so you have know it yourself where each chunk is located.

In order to simplify this, a helper functionality is included which creates a box layout of chunk offsets procedurally. Simply define the boundary in the Box Layout X, Y, Z and hit Create Box Layout and it fills the Chunk Offsets list procedurally.

For example 3,3,3 would create 27 entries.

VOXEL COLLISION MODIFIER

This component can be used on RigidBodies to modify a voxel map when the RigidBody collides with it. The main parameters also are **Depth, ID and Carve Radius**. The target VoxelGenerator is fetched automatically during collision.



Also the **positioning** can be set to center point or collision point. Center point will apply modification where the center of the RigidBody is used as center point. Collision Point will use the impact point calculated by the collision detection system as center point.

Max Contacts are used to determine how many contact points are used for modification.

It is also possible to limit the possible target generators by using whitelist When **Use White List** is true, a list appears where you can assign VoxelGenerators which should be modified by the RigidBody.

Mode can be used to define the modification type as it is seen in other modifiers.

It is recommended to keep the amount of “Hot Objects” low since the whole process is very expensive. The sample scene MeltDown shows how a gold sphere is melting through the snow.

VOXEL UTILITY:

The main function provided by the Voxel Utility is the **ModifyVoxels** function and comes in 2 variations: One function for spherical and one for box shape modifications. The main input parameters for the standard **ModifyVoxels** function are:

- generator: the target VoxelGenerator:
- Center: center point of the modification in world coordinates
- Orientation: rotation of the modification
- Size: box size of the modification
- Radials: Vector4 parameter for radial limits: X, Y, Z is the radius for axis aligned circles and can be used to obtain cylindrical modification shapes. The last value W is used as spherical radius.
- Depth: target depth
- ID: ID of the modification
- Mode: Modification mode.
- Dimension: Target Dimension if voxel generator has more than one dimension.

Modes: There are separate modes for left and right click including a separate target ID applied.

1. Set: Sets the voxel ID to the target ID.
2. Additive: Adds the target ID to the voxel ID (target ID can be negative)
3. Subtractive: Subtract the target ID to the voxel ID (target ID can be negative.
4. Smooth: Applies mean filter to the target region (ignores target ID)
5. Rough: Unsmooth target region (128 becomes 255, 127 becomes 0)
6. SolidPaint: Changes voxel ID to target ID if voxel ID is not zero (for AtlasCubes painting)

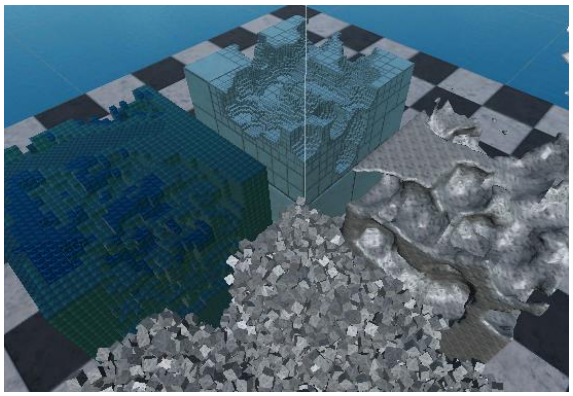
The spherical variant of the **ModifyVoxels** function has the same input parameter except size and radials as it is replaced by an radius parameter.

The box variant has the same input parameters except the radials as it is not needed.

Additionally, this helper class provides functions to obtain information about the modification before it starts. **EvaluateModificationCount** will calculate how many voxels will be modified. This function only needs the generator, radius or size and target depth as input parameter.

Before modifying anything, it is recommended to check, if the modification is save. Therefore **IsSave** function is provided which also requires generator, radius or size and target depth. This function will return true when the amount of voxels which will be modified is greater than the included safety limit of 100000 (you can change it on your own risk).

HULL GENERATORS



Hull generators are responsible for the visualization of the voxel map and must be attached as children to VoxelGenerator GameObjects. Each hull generator has its own settings such as resolution and material and any number of hull generators can be attached to the VoxelGenerator. This allows the generation voxel blocks with very unusual properties.However also keep in mind that more hull generators result in a slower update rate when the voxel map is modified as there is more work to do.

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

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.

If **Locked** is set, the hull generator will not update his content.

If **Lock after init** is set, the hull generator will only update its content when the VoxelGenerator is initialized. Afterwards the hull generator will be locked until the VoxelGenerator is cleaned.

IMPORTANT UPDATE!

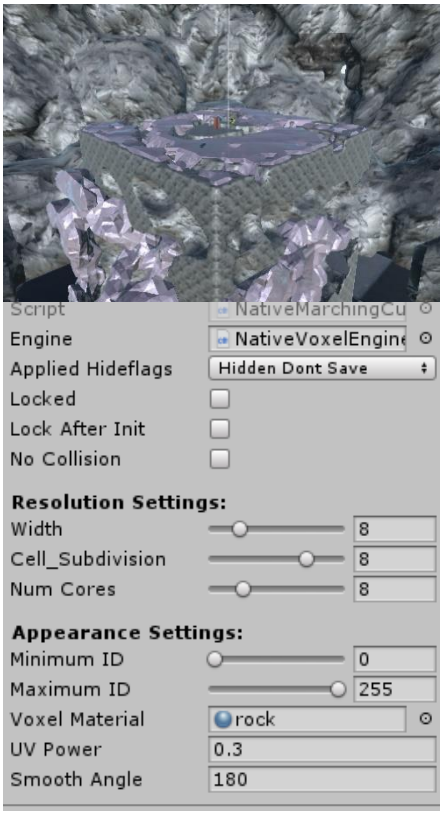
Over the years, the amount of hull generator scripts grew to a chaotic amount due to all the performance optimization phases. As performance improvements came over time, there are now many derivates of marching cubes on GPU and CPU. Also some have other weird features such as "Scientific" or "Experimentel" where the data remains on the GPU. So there is quite a lot of chaos regarding hull generators right now.

In order to solve this, there is now a hull generator called "ModularUniformVisualHull" which consolidates all the scripts scattered around into one modular hull generator.

This means that long term it makes no sense to keep inferior hull generators which means that most will be deprecated and removed when Voxelica V2 is released.

In future, having just one Hull Generator greatly reduces maintenance overhead for me and it is easier for customers. Below is a detailed section dedicated to the ModularUniformVisualHull hull generator

Marching cubes is the most common visualization method for voxels and is mostly used for terrain or other natural environments because the appearance is completely smooth. The smoothness is caused by the ID value which is between 0 and 255 so even voxel maps with low resolution have smooth surfaces.



The first settings are the resolution settings. These are common in other hull generators and define the resolution of the final geometry. Changing most settings while the VoxelGenerator is initialized will trigger rebuilding the hull generators.

Width: Defines the resolution of each mesh piece. For example a width of 8 means that one piece contains 8x8x8 or 512 voxels.

Cell_Subdivision: Defines how many mesh pieces will be generated. For example a value of 8 means that 512 mesh pieces will be generated where each mesh piece will cover 512 voxels if width is set to 8. The total voxel amount covered by this voxel block would then be 512x512 (262144) voxels.

NumCores: This value defines how many mesh pieces are processed each frame when the hull must be updated. The works of every mesh piece are evenly distributed along the CPU cores (is decided by the Unity Job System)

The appearance settings are specific for each type of hull generator and define the visual appearance. It usually contains a material, UV power and optional smoothing of vertex normals.

MinimumID/MaximumID : The ID range covered by marching cubes**.** Solid geometry starts to appear when the ID value of a voxel is in the upper half of the covered ID range. For example solid geometry begins at 128 when the default range of 0-255 is used.

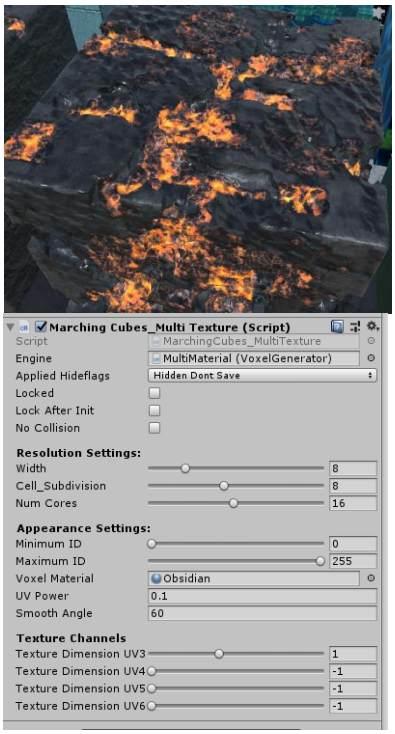
VoxelMaterial: Material used for the mesh pieces.

UV Power: UV Multiplier for the mesh pieces

Smooth Angle: Smoothing Angle of the vertex normals.

MULTI TEXTURE SUPPORT (ADVANCED)

Since Update 1.5 Multi Texture is possible. In order to use this feature, use MarchingCubes_MultiTexture instead. Also the dimension count of the VoxelGenerator must be two or more. The first dimension is used to describe solid or non-solid and the additional dimensions are used to write into the respective UV coordinates. The texture channels define which UV set should be affected by which dimension.



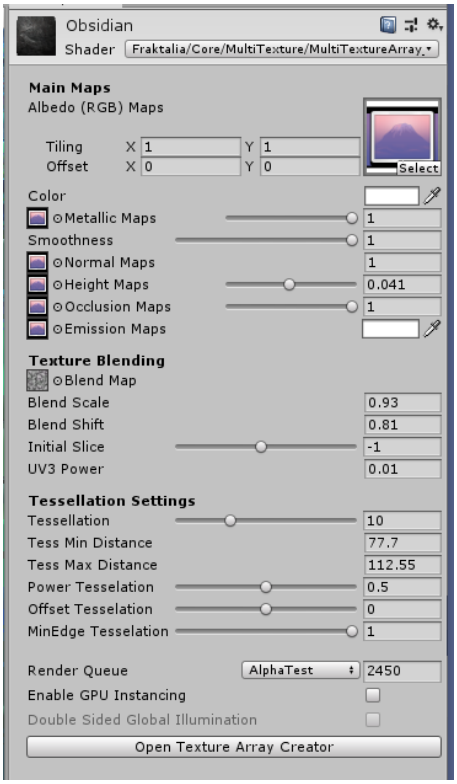
In the example image, the second dimension writes to the UV3 coordinate of the procedurally generated mesh. The shader of the assigned material uses the UV3 value to define which texture should be applied. The material uses a specific shader which uses Texture2DArrays instead of normal textures.

The simplest shader included in the core library reads UV3 in order to determine the blending value.

Also this hull generator uses the vertex color array of the mesh as barycentric coordinate which is required to implement 100% seamless tessellation.

USING TEXTURE ARRAYS

The included core library contains such shader which replicates the Unity3D Standard shader but with tessellation and multi texture arrays. Most settings are almost identically to the standard shader.



Texture Blending contains settings which define how to blend between texture slices inside the assigned texture arrays. A blend map can be assigned in order to add some variation to the blending. **Blend Scale and Blend Shift** describes the influence of the Blend Map.

Initial slice defines the initial blending state.

UV3 Power describes how the UV coordinate influences the blending state. Since possible values are between 0 and 255, this value is usually very low (1 / 256).

The Tessellation settings are used to further enhance the voxel representation even on low resolution voxel maps. The first parameter is the tessellation strength where higher values subdivide the mesh the most.

Min/Max Distance: Parts of the mesh which are further away from the camera are not subdivided in order to save GPU power. Parts below the minimum distance are subdivided the most.

CREATING TEXTURE ARRAYS:

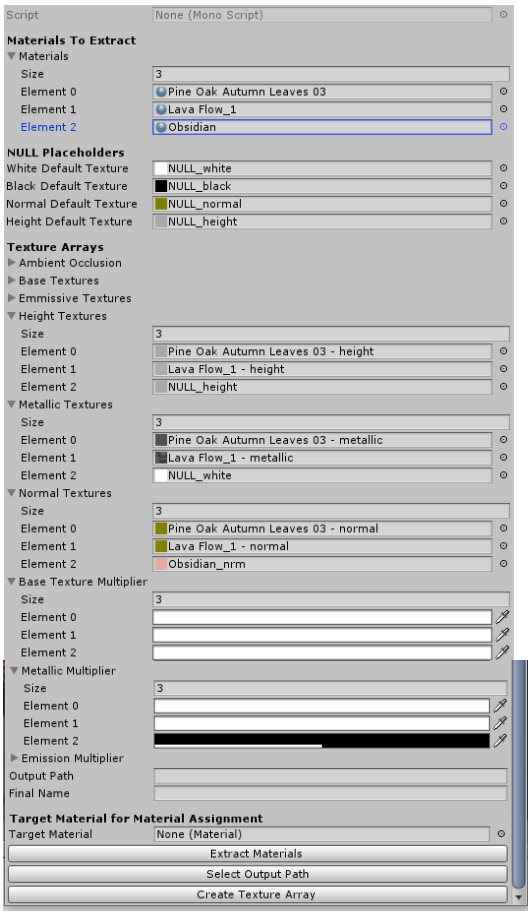
Unfortunately, Unity3D does not provide a user friendly way to create texture arrays. Therefore when clicking the “Open Texture Array Creator”, a custom editor window will popup which allows you to create Texture Arrays.

This tool can extract the textures used in selected materials which use the standard shader and bakes those textures into texture arrays. If the material does not use the standard shader, Null textures are used instead. The perfect materials are Substance materials.

Color arrays will also be multiplied into baked textures since every material could have different Metallic, Emission and Main Colors.

However it is also possible to assign materials manually by simply filling the arrays.

Then you can select the output path and name. If the output path is not valid, the root asset folder is used instead. You can assign a target material which supports texture arrays and the created texture arrays are assigned automatically for you.

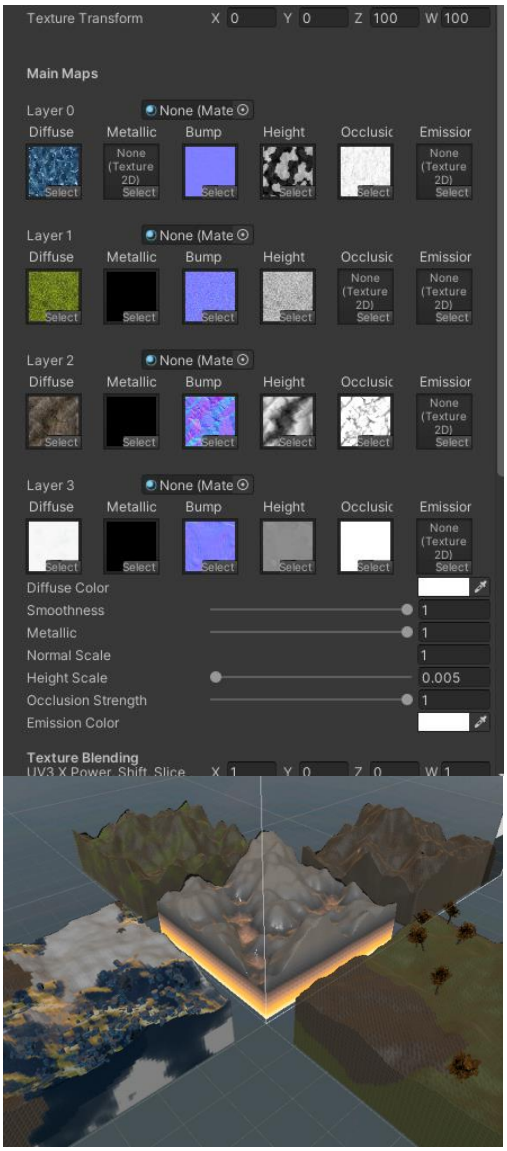


USING NORMAL TEXTURES:

Texture Arrays have the disadvantage that they are not compressed and it is not easy to swap out individual slice. This shader can be found under:

- Fraktalia/Core/MultiTexture/Texture2D_UVBlend_Triplanar.

In the end you simply assign the individual textures for each layer. Functionality wise it is identical to texture arrays except that it is fixed to 4 layers which is a limit caused by the Sampler States. More layers may be possible in the future when GPUs improve



You can also apply a “Standard” material by filling in the slot. Then a button appears which automatically assigns the textures of the Standard material so you don’t have to fill in each texture manually.

Important Note:

Not all Mobile Devices and support SamplerStates. WebGL is not supporting this at the time this was written.

Your target hardware must support Direct3D 11/12 or Metal according to the official documentation:

https://docs.unity3d.com/Manual/SL-SamplerStates.html

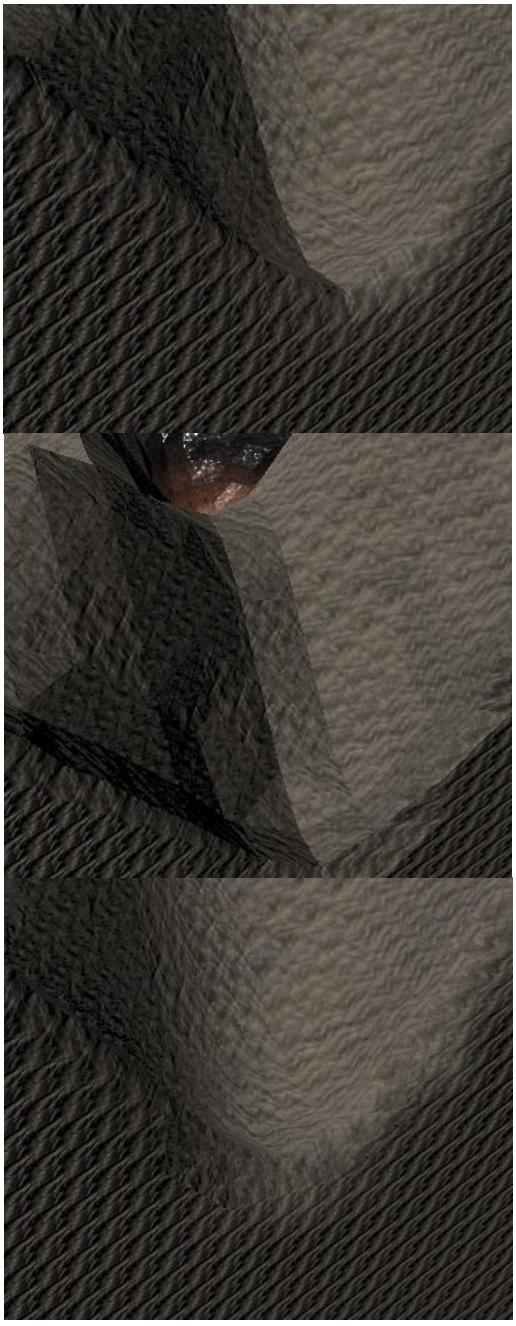
NEW CALCULATOR

Marching Cubes has now a new option to use an improved calculator which derives the vertex normal from the voxel map.

The vertex normal from the standard normal calculation has artifacts between cells when the smooth angle is greater than 0 (Top Image). Also the calculation of smoothed angles is very expensive when the mesh itself is used to calculate those normal.

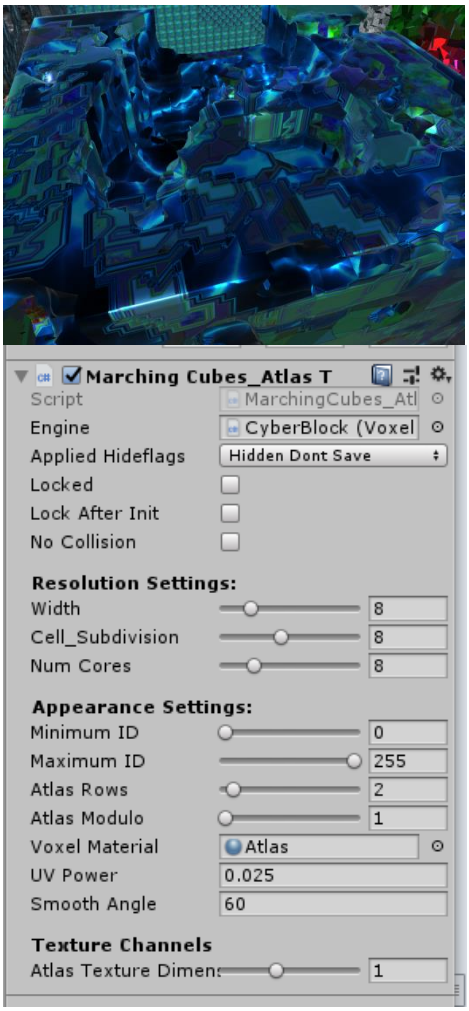
Flat shading (Middle Image) does not have these artifacts as they are calculated from each face and is cheap to compute. However it only makes sense in low poly environments.

The bottom image shows the result of the new normal calculator which is seamless and perfectly smooth. The only drawback is the imperfect Cube-Based UV coordinates. Therefore using a triplanar shader is highly recommended.



TEXTURE ATLAS SUPPORT

While the first multi texture support creates natural locking results since it allows seamless blending between multiple textures, this method uses a conventional texture atlas which cannot create seamless transitions. Since this method only modifies the UV1 coordinate for the texture atlas, this method does not require special texture array shader.



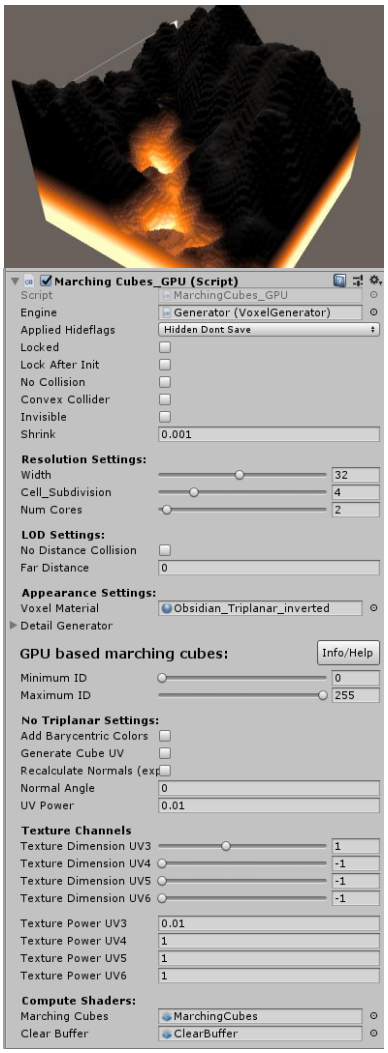
The disadvantage is that smooth blending between textures is impossible. This gives the result an artificial and tech-like result. The parameters are almost identically to the normal marching cubes hull generator.

Atlas Row defines the subdivision of the atlas texture. For example if the atlas contains 4 textures, the row count is 2.

Atlas Modulo is used to define vertex skips and alters the final appearance. Nice values are 0, 3 and 6.

The last parameter is the **texture dimension** which defines the voxel dimension for the texture evaluation.

GPU BASED MARCHING CUBES



All components of the Voxel Generator are using the CPU for calculations while the GPU is only used for the traditional rendering. Modern GPUs are very powerful and their computation power can be used for hull generation. Therefore this hull generator now combines CPU and GPU in order to maximize computation speed.

The main disadvantage is that this hull generator only works on hardware which supports Compute Shaders. Check page provided by Unity3D if your hardware supports compute shaders:

https://docs.unity3d.com/Manual/class-ComputeShader.html

The settings are identically than the normal CPU based Marching Cubes but is Triplanar by default. The vertex normals are directly calculated using the Voxel Map and allows perfect Tessellation.

The No Triplanar Settings allow the extra calculations of desired properties at the cost of extra computation time. Especially the recalculation of normal based on the mesh (method used by all other hull generators) is very expensive.

Also by default, no UV1 coordinates are used as it is highly recommended to use the now included triplanar shader but it is possible to generate the Cube-Based UV coordinates like in CPU based Marching Cubes.

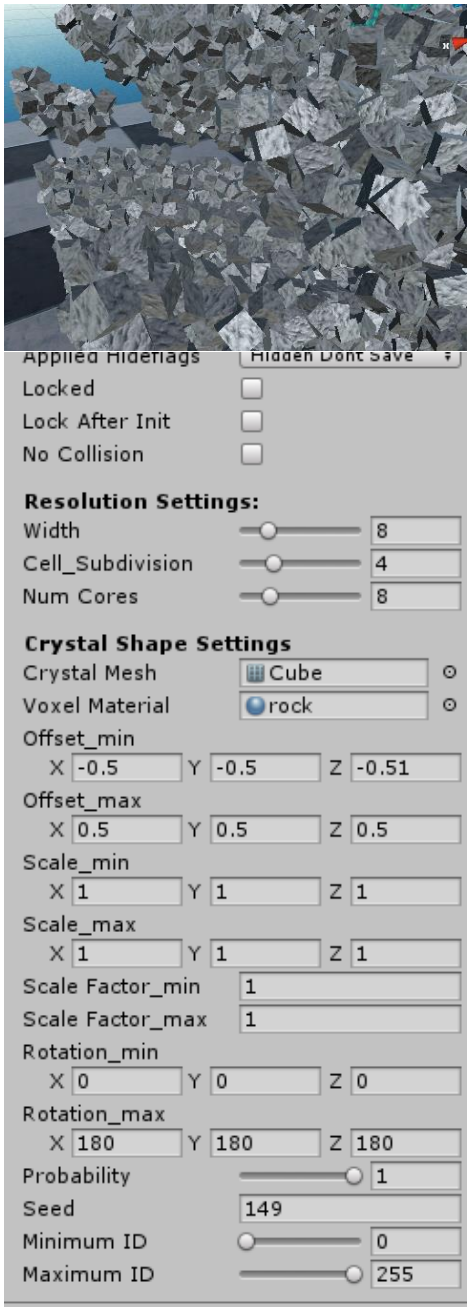
The last properties are the compute shaders which can be assigned but are loaded for you if they are not assigned.

UPDATE 1.9.5

The time where the GPU is faster than the CPU is over! The introduction of MarchingCubes CPU V2 manifests the end of GPU based visualization systems as the CPU is now faster than the GPU version by utilizing optimization techniques not possible on the GPU!

CRYSTALLIC

This hull generator generates a crystalline structure similar to the structure found in the Crystal Generator asset. 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.

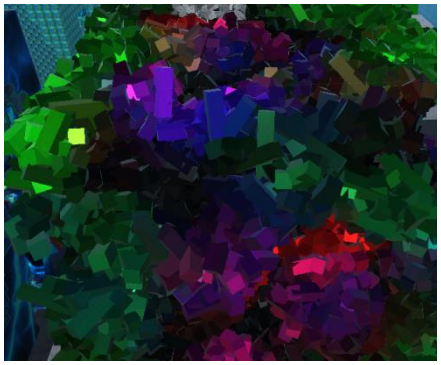
Rotation Min/Max: Random rotation of each crystal

Probability: Probability of crystals being generated for each voxel creating a surface.

Seed: Seed used for randomness.

Min/Max ID: Similar to marching cubes and used to determine surface regions.

MULTI COLOR CRYSTAL (ADVANCED)



This variation implements multi colored crystals and has the same properties as the crystal hull generator. It has extra channels which modify UV3-UV6. This shader also requires a special shader which uses those extra UV coordinates. The core library includes a standard surface shader which uses UV3-UV6 as color values.

This hull generator can also be used to create crystals with different materials using texture arrays such as the marching cubes variations. In the end, the assigned material decides the purpose of the extra UV coordinates.

ATLAS CUBES

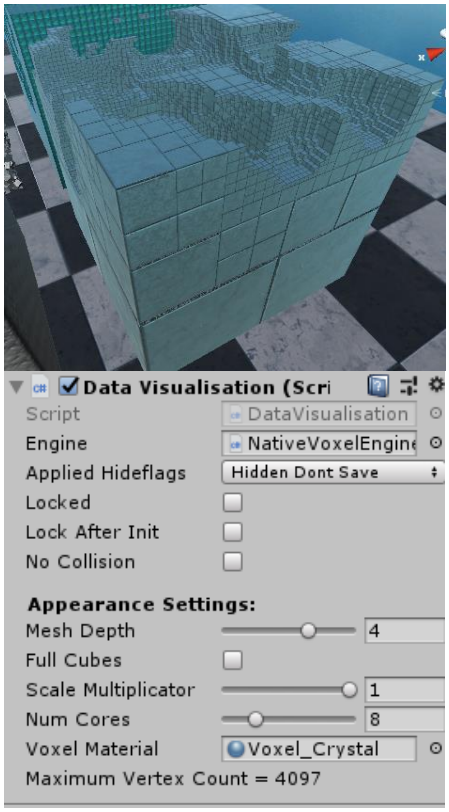
Atlas cubes is the typical hull found in Minecraft where ID has its own UV coordinate assigned. This allows the visualization of 255 different blocks. The original sprite sheet is a 2048x2048 texture which is subdivided into 16x16 cells. The current method in Minecraft uses strings which are not supported by the Unity Job system yet.

This hull generator also has the resolution settings found in the previous hull generators. The only new setting is the **Atlas Rows** which defines the cell subdivision of the sprite sheet.

The cheap sprite atlas provided in the sample has 2 rows and 2 columns. Therefore the Atlas Rows is set to 2.

Voxel Material and Smooth Angle are identical to the previous hull generators.

DATA VISUALIZER



This hull generator works completely different than the previous hull generators. Instead of covering a fixed region of voxels, this hull generator directly converts the voxel map into a visual representation. Voxel ID greater than 0 is visualized as solid block. Also this generator has no resolution setting and the main settings related to the appearance only.

Mesh Depth: Describes the complexity of each mesh piece. Lower value means more GameObjects are required to represent the voxel map. Higher value mean that less GameObjects are required but the vertex count of each mesh piece is higher. This setting also includes a safety check especially when the subdivision power of the VoxelGenerator is greater than 2.

Full Cubes: Walls which are obstructed by solid voxels are usually not generated in order to reduce vertex count. If true, this is suppressed and therefore only useful when the scale multiplier is smaller than one.

Scale Multiplier: Reduces the scale of individual voxel blocks. The result can be used to generate fancy effects.

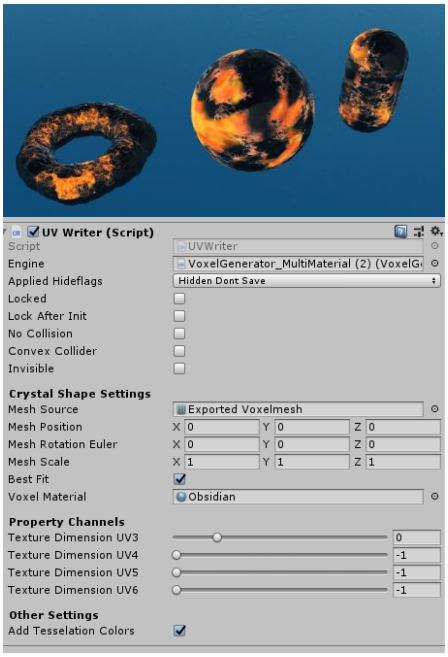
NumCores: Update speed evened out on multi core CPU as seen in other hull generators.

VoxelMaterial: Applied Material

IMPORTANT NOTE:

This visualization method is deprecated and will be removed in Voxelica V2 because it is slow and quite buggy!

UV WRITER



The UV Writer is a hull generator which manipulates the additional UV coordinates of an assigned mesh shape. The sample shows the default sphere, capsule and a exported mesh which are modified using the UV Writer.

This hull generator makes only sense if the additional UV channels 3-6 are utilized by the material. Therefore custom shader who use those channels are mandatory (or you will see no effect).

Mesh Source: The reference mesh which should be modified. It is important to know that the result is a new mesh which is generated during initialization. Therefore the original mesh is never modified. Use the mesh exporter from the VoxelGenerator if the result should be saved.

Mesh Position/Rotation/Scale: Defines how the mesh should be positioned inside the volume boundary.

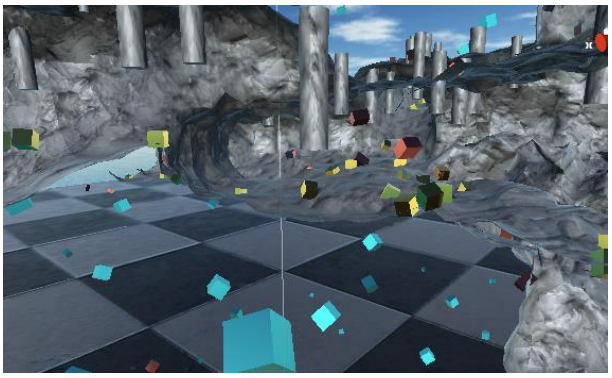
Best Fit: The mesh is positioned inside the volume boundary so it fits perfectly while keeping aspect ratio.

VoxelMaterial: Applied Material

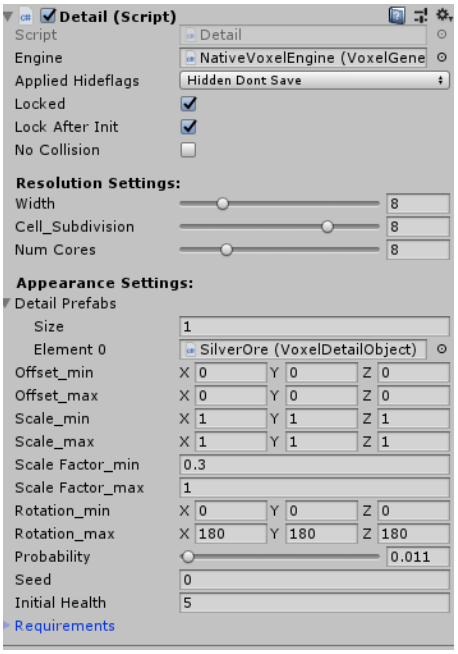
Property Channels: Define which voxel dimension should be written into which UV coordinate. A value of -1 will write nothing into the affected UV coordinate.

Add Tesselation Colors: When true, barycentric coordinates are written into the vertex color channels which are important when using wireframe materials or seamless tessellation features.

DETAIL



This hull generator is the most complex generator and allows the placement of detail objects into the interior of the voxel map. Any GameObject can be assigned as Detail object which will then be placed over as it is shown in the right image. Silver cubes are generated inside the solid volume. Gold Ore is only generated on the ground surface, Copper on any surface and Stalagmite Cylinders on the ceiling. The placement behavior is defined by a variable amount of requirements. Every detail at a voxel position can only be generated once. Also for ores which should be dig out, it is recommended to set LockAfterInit to true else ores may spawn while the player is digging where some voxels meet the requirements which were not met before and place details. Sometimes you actually want this feature for example the randomly placement of toxic gas on the surface when the player removes material.



The Detail hull generator also has the resolution settings as found in previous hull generators. The most important parameter of the appearance setting is the **Detail Prefabs** list where you assign any number of GameObjects which have a special VoxelDetailObject script as component.

When a voxel meets the requirements and the probability also matches, a random object from the Detail Prefabs list is chosen and a instance is placed at the voxel coordinates.

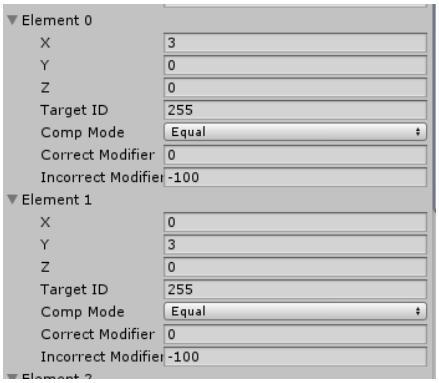
The position, rotation and scale of placed detail objects are further manipulated by:

- Offset_Min/Max
- Rotation_Min/Max
- Scale_min/max, Scalefactor

The **probability** parameter decides if a detail is placed even when requirements are met. It is generally recommended to keep the probability low as the amount of GameObjects could massively increase if the requirements are not strict enough. The randomness is completely deterministic and is based on the **seed** parameter.

NOTE: THIS HULL GENERATOR IS NOT SUITABLE FOR INFINITE WORLD SYSTEMS

REQUIREMENTS



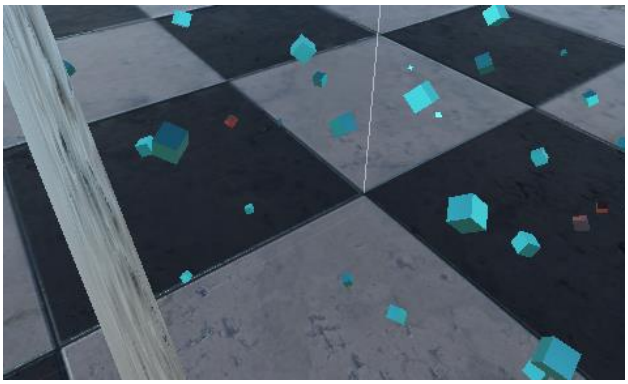
The requirements are used to evaluate if the environment of a voxel has specific values and fit given specifications. The Requirements list can contain any number of detail entries where the **InitialHealth** is the main starting parameter.

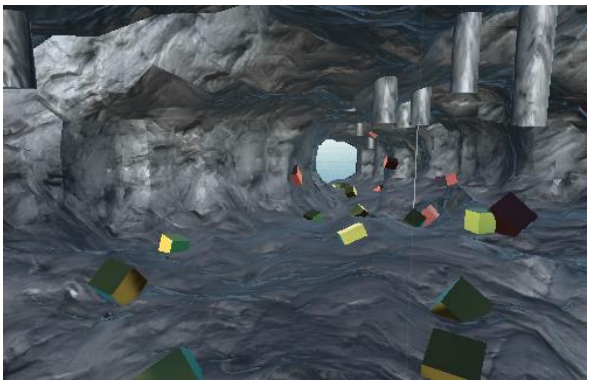
When a voxel is evaluated, the InitialHealth is the starting parameter. A voxel meets the requirements if the InitialHealth is greater than 0. The right sample entry checks the ID of a voxel which is 3 blocks away from the evaluating voxel. If the ID of the neighbor is not 255 or in other words completely solid, the health will drop to -100 invalidating the evaluating voxel immediately. If position X, Y, Z is 0, the ID of the evaluating voxel is checked. This sample has 4 more entries which also checks the neighbor for the 4 other directions and the resulting effect is that details can only be placed when surrounded by solid geometry.

Detailed Parameter Information:

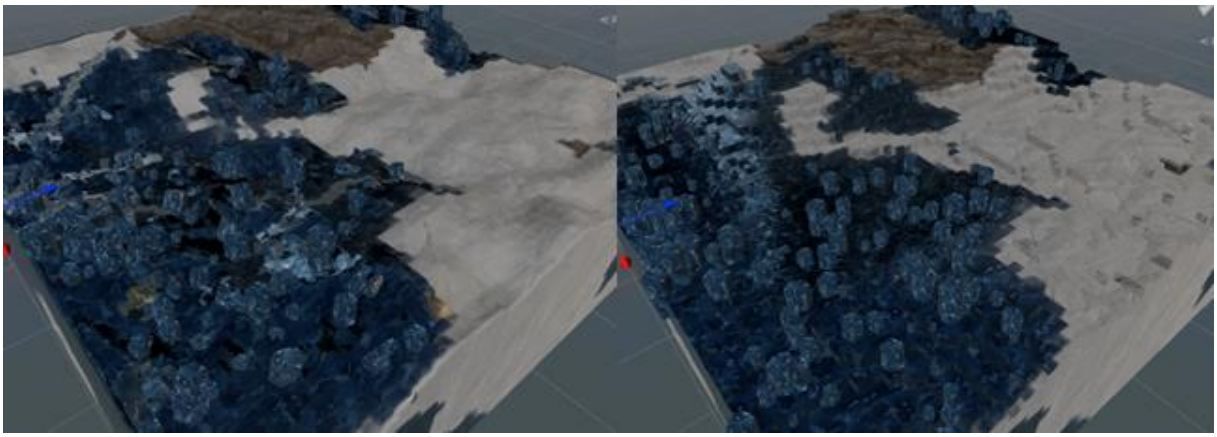
- **X, Y, Z:** Coordinate of neighbored voxel which should be checked. If zero, the voxel which should be evaluated is checked.
- **Target ID:** The target ID which a neighbored voxel should be compared to.
- **Comp Mode:** Comparison mode. Possible options are Equal, Not Equal, Greater and Smaller.
- **Correct Modifier:** Changes in health if this specific requirement is correct.
- **Incorrect Modifier:** Changes in health if incorrect.

The sample scene “Detail” shows what is possible. It is possible to place objects inside the solid volume like in the left image where silver cubes are placed but it is also possible to place objects on the surface only like in the right image. In the right image, gold ores will only spawn on ground while stalagmites can only appear on the ceiling.





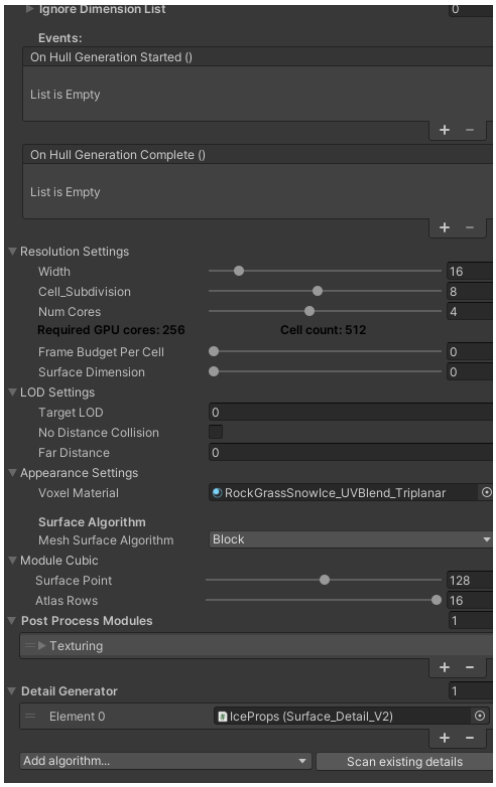
MODULAR UNIFORM VISUAL HULL



This hull generator introduces a module based visualization system. When using previous hull generators, switching between algorithms like MarchingCubes and Cubic meant removing the MarchingCubes script and attaching the Cubic script which is tedious. With this hull generator, all you need to do is changing the Mesh Surface Algorithm.

You can switch between None, Marching Cubes CPU, Marching Cubes GPU, Block, Crystal, Mesh(UV Painting).

The settings are similar to the previous hull generators. Also this hull generator allows you to define which dimension it should use for the surface calculation.



LOD Settings:

It fully supports LOD which is defined by the Generator. If “No Distance Collision” is checked, collision hull will not be calculated if the LOD from the Generator is higher than the Far Distance.

Target LOD defines when the resolution should be lowered. If the current LOD from the Generator is higher than the Target LOD, the width is cut in half.

Appearance Settings:

Here is everything related to the Material and Geometry algorithm used.

Keep in mind that the multi material functionality is now a post process module!

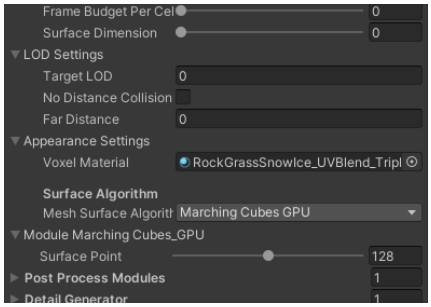
Post process settings:

Here are modules for further manipulating the resulting geometry.
Keep in mind that the multi material functionality is now a post process module!

Detail Settings:

This section is for assigning Surface_Detail_V2 which is for foliage and props placement.

SURFACE ALGORITHMS

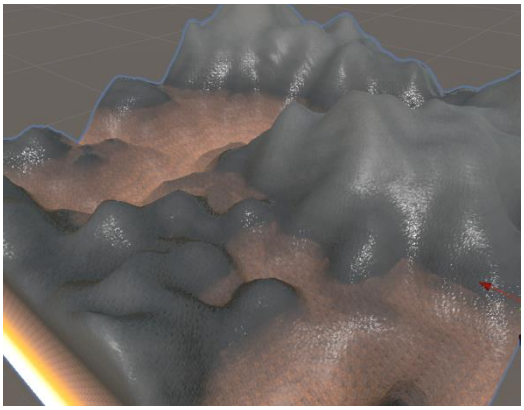


Surface algorithms calculate the surface geometry and every MUVH hull generator has one assigned. It is located in the Appearance Setting.

There you can select the surface algorithm from a dropdown menu. Most common surface algorithm used is either Marching Cubes or Cubic.

Keep in mind that not all surface algorithms produce UV coordinates! UV coordinates are now created using post processes.

MARCHING CUBES



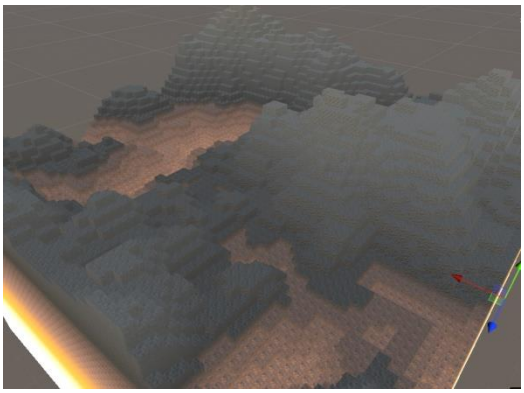
This is the default setting and has a CPU version and a GPU version.

It is recommended to use the CPU version as it is faster now unless your GPU is so much more powerful than the CPU.

Overall the GPU should stay with rendering as countless optimization sessions have shown that the GPU is quite terrible with voxel unless the whole data permanently resides on the GPU.

Does not create UV Coordinates!

BLOCK



This is your standard “Minecraft” like rendering where each voxel is displayed.

As usual, 0 means completely empty and all other values are considered solid. But by default the surface point is 128.

Additionally it automatically fills the UV coordinates according to the Voxel ID so a texture atlas can be used.

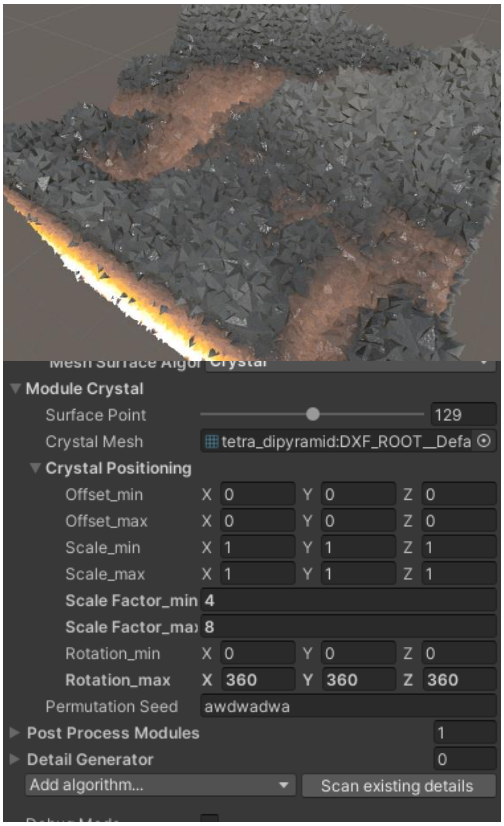
Atlas Rows is by default 16 so it is compatible with a 16x16 texture atlas.

CRYSTAL

Crystal is one of the more unusual visualization modes for Voxel and is pretty unique as it creates the surface using crystals.

This surface algorithm requires a crystal mesh which is used as template in order to create a surface of crystals.

Additionally you can define positioning settings to introduce randomness. The seed is used for creating the permutation table since randomness must be pre-created during initialization.



MESH

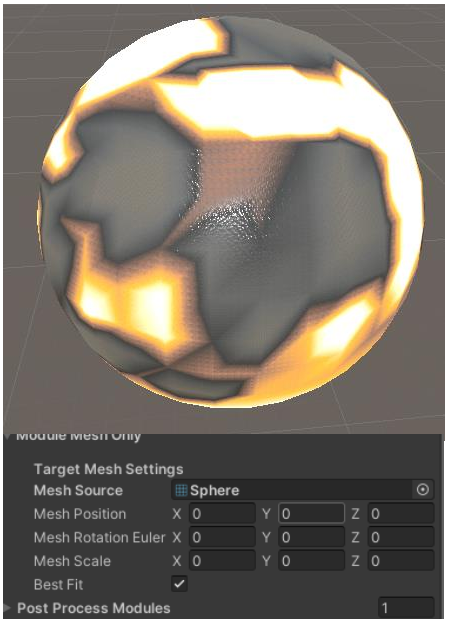
This surface algorithm allows you to paint the surface of an existing mesh.

In order for this to work properly, Cell Subdivision and Num Cores must be set to 1 as it has no “cells”.

After you assigned the Mesh Source, the mesh will be used in order to define the surface.

The mesh will be positioned, rotated and scaled according to the settings. If best fit is checked, it will be placed into the volume ideally without distorting the mesh.

The main usage for this is if you have created the geometry using other methods outside Voxelica and you just want to manipulate the mesh according to the voxel data.



POST PROCESS MODULES

The post process modules contain all the small features scattered around all hull generators. This also includes surface modifiers which could simply become a module instead.

These modules manipulate everything else which can be done after the expensive surface calculations! For example manipulating UV3 coordinates for multi texturing can be done independently of the surface algorithm.

This means if you modify Dimension 1, only the post process modules have to be recalculated as the solid surface itself remains the same. Since the surface calculation can be skipped entirely, manipulating everything else except the Dimension used for density is magnitudes faster.

See technical details section for more information.

MULTI TEXTURE

This post process module writes into the UV3, UV4, UV5 and UV6 coordinates.

Since X and Y coordinates are written separately, you can store up to 8 voxel dimensions into a mesh.

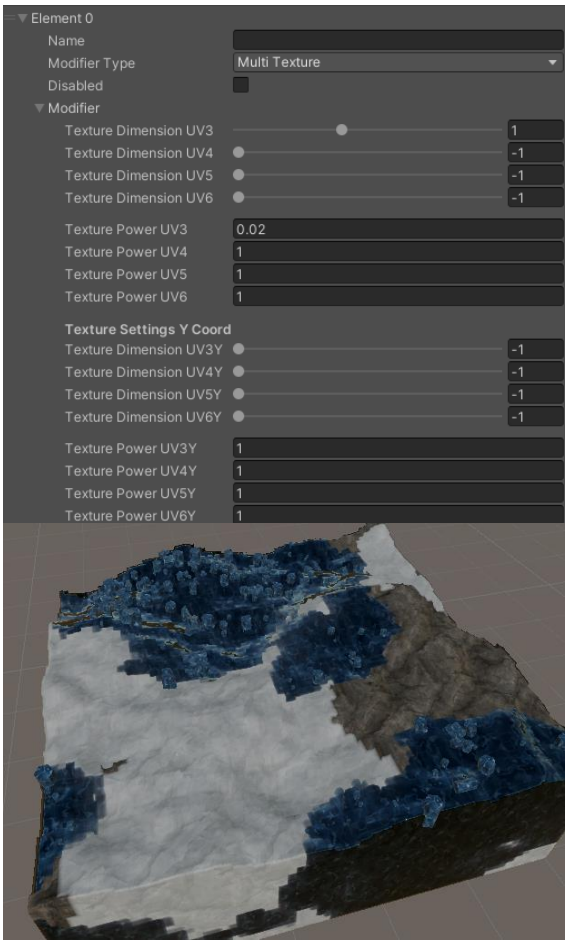
Shaders then can use the information in those UV coordinates in order to manipulate the rendering.

The most common one is texture lookup where the UV3 X coordinate is used as index lookup. Combined with a smooth blending gives you the resulting material seen in the samples.

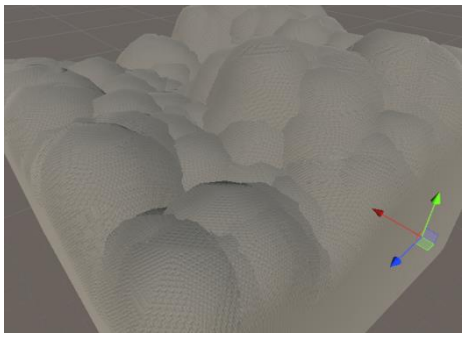
Another example is using 4 Dimensions where one is used for each texture like in the right image. The shader itself has 4 textures and the UV3-4 values define the stength of each texture.

Your shader could use the full range UV3-UV8 but not every target platform supports a higher UV range on a mesh.

Note: UV0 is used for texturing in Unity. UV1 and UV2 are also reserved by Unity.



INFLATE

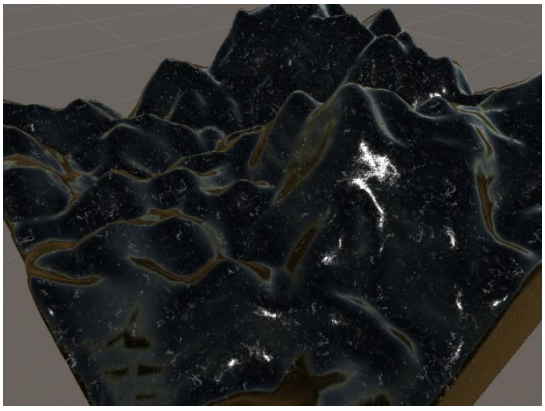


This post process module inflates or deflates the mesh.

There are some usages where you need this and this is mostly related to navigation meshes and other weird processes.

Not suitable for visualization as it obviously looks ugly.

CUBE UV

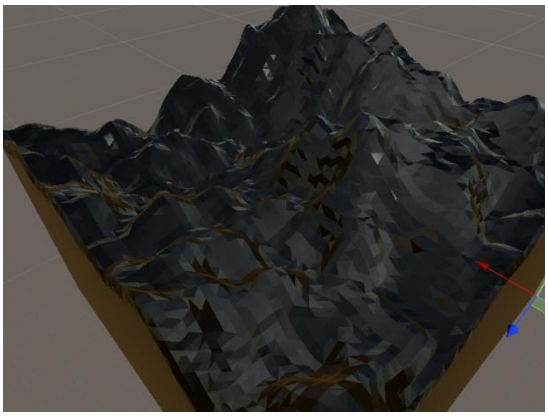


By default, the hull generator expects you to use triplanar mapping where no UV coordinates are required.

If your shader needs UV coordinates as it does not use triplanar mapping, apply this post process otherwise you won't see any texturing

Using non-triplanar mapping may not be as "perfect" as triplanar mapping but it is much cheaper to render, making it suitable for weaker GPUs

NORMALS



By default, the normal are derived by the surface algorithm.

However under some circumstances you want flat normal and usually is desired a low-poly art style.

It can also recalculate tangents

CALCULATE TANGENTS

This post process simply recalculates the tangents. Sometimes this is required if the normal mapping is wrong.

VERTEX COLORS

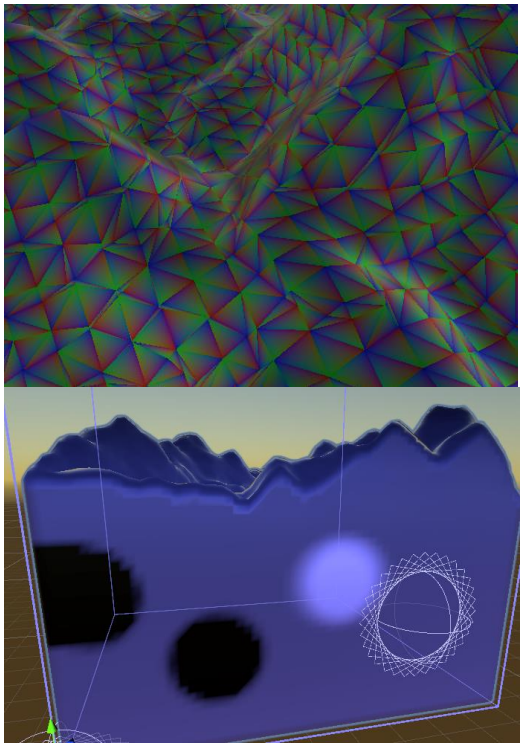
Besides UV3-UV6, every mesh has a vertex color which can be used for rendering if UV3-6 is not enough. This gives you additional 4 dimensions you could write into the mesh.

Additionally, this post process can write barycentric coordinates into the vertex colors

Barycentric coordinates are required for wireframe shaders and seamless tesseleation. The upper right image shows the result if such coordinates are written into the vertex colors.

Second right image show the normal usage when dimensions are written into the vertex coordinates.

There are shaders included which display the vertex colors as the standard shader does not do that.

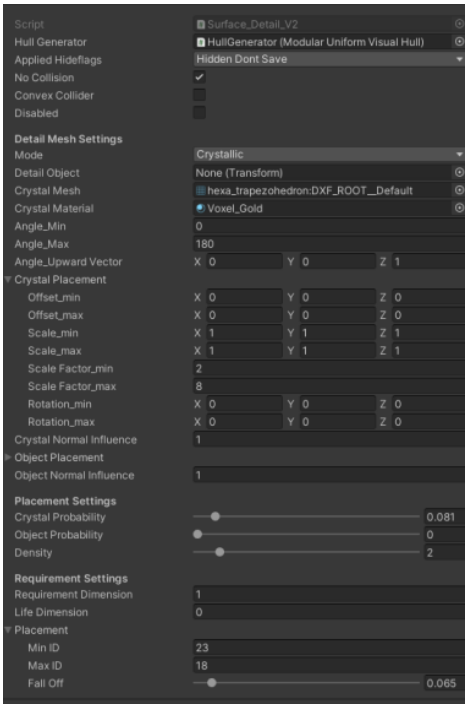


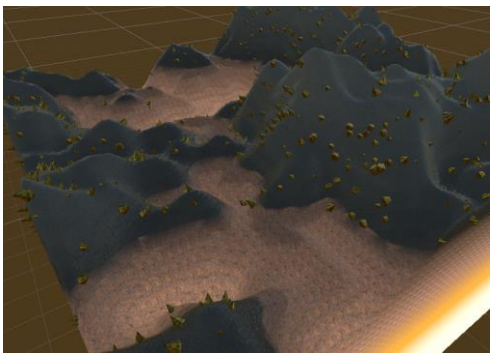
DETAIL GENERATORS

The last functionality is the application to place foliage and props onto the surface. Once added, a new separate object is created which has “Surface_Detail_V2” script attached.

See surface detail below as the only difference is the massively increased performance!

Once you have added the Surface_Detail_V2 object, cleanup the engine and reinitialize it in order to have it applied.

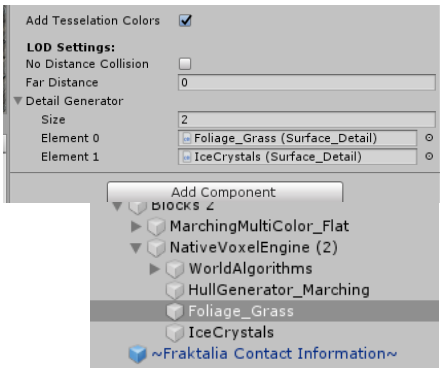




SURFACE MODIFIER

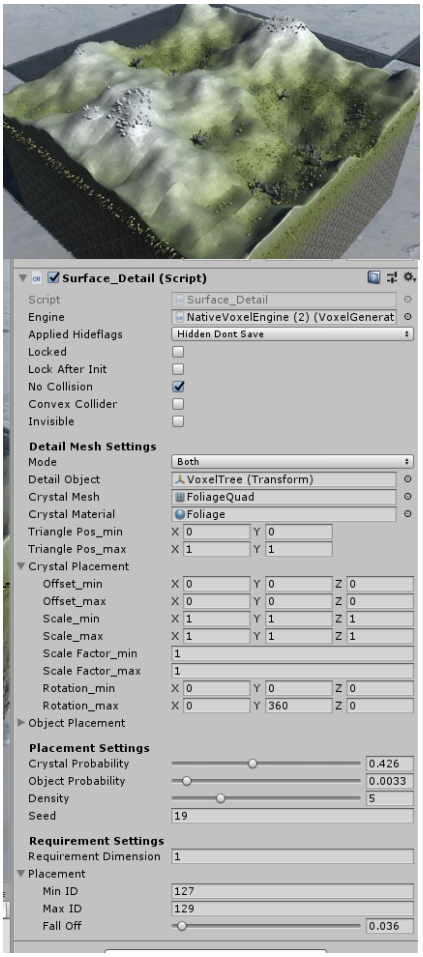
Surface modifiers alter the surface generated by a hull generator. Currently only Marching Cubes multi texture supports surface modifiers.

They inherit from Basic Surface Modifier and are applied after the hull generator finished hull generation. A hull generator supporting surface details has a Detail Generator list which can be filled with multiple detail generators.



SURFACE DETAIL V1, V2

Surface Detail is a modifier which places detail objects to the surface. Mostly Details should have no collision especially if the detail is foliage like the sample grass. The surface detail generator has 2 modes which is Crystallic and object mode.



Crystallic is recommended for details with low vertex count such as grass sprites or small pebbles. It uses the Crystal Mesh parameter as shape and is a simple quad and a crystal material.

The crystal placement provides position, rotation and scale settings for variation.

Object placement is recommended for complex objects such as trees and props with more vertices. It places instances of the Detail Object and uses the Object placement for variation.

Crystal and Object have both a Probability. Crystals are also multiplied by the Density as more crystals are possible as the performance impact is lower than with highly detailed objects such as trees.

The Seed is used as initial parameter for randomness in order to keep the result deterministic (else the props would jump around rapidly)

The last settings are the requirements to define when a detail should be placed. You can define which dimension to read and the ID the voxel at the surface should have to be suitable for placement. The sample restricts the placement of grass to a region which contains grass. (127-129)

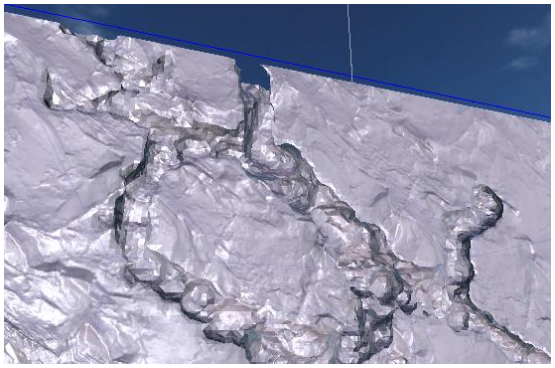
The falloff defines how sharp the probability is reduced when the surface is outside the desired ID range. Also the right sample uses Both as mode which places crystals (grass) and objects (trees).

V2 Is identically but is significant faster!

MOBILE SUPPORT:

VoxelGenerators can be used on any mobile device. However it is recommended to use lower resolution voxel maps and less detailed visual hull constructions since mobile devices are not as powerful as modern gaming computers. The recommended mesh resolution for marching cubes is a width of 4 and cell subdivision of 4. The main limitation is GPU based because the VoxelGenerator is optimized for multi core CPU usage since the GPU already is busy rendering the geometry.

VR/HIGH END SUPPORT:



VoxelGenerators can be used in VR applications. Actually the development of this asset was part of the application for my master thesis which was about sculpting in VR. There it was possible to sculpt the voxel block using nasty power tools like chainsaws and concrete cutters. Back then the voxel engine was much less optimized than it is now.

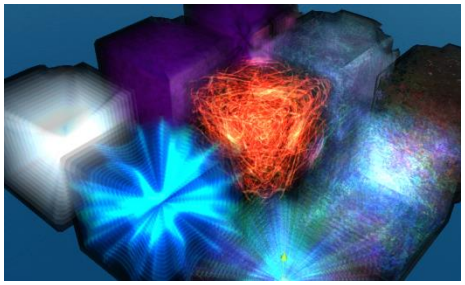
Creating a sculpting tool for VR is done by simply adding a collider to the handle with a voxel modifier and call ModifyAtPos using the collision point(world position) as input parameter.

For high end stuff, a resolution where the width is 16 and sell subdivision is 16 already is fine enough where individual voxels start to blur and become not visible anymore. Going smaller like width of 32 and cell subdivision of 32 is so fine that it enters the realm of scientific visualization where no one cares if it works in real time or not (and where money doesn't matter).

ASSET EXTENSIONS

This section contains information about extensions included in other asset packages. An asset extension combines the Voxel Generator with a different unity asset. The simplest example would be marching cubes using a gemstone material from the Advanced Gem Shader package (costs 15$ or two unhealthy McDonalds meal). Expansion assets are .UnityPackage files which are located in the extension folder of the other asset. The extension content will not work properly if the Voxel Generator is not included in the Project. For example if you import the extension from the Advanced Gem Shader asset, the sample prefabs simply have “Missing Scripts” as the VoxelGenerator doesn’t exist.

ADVANCED GEM SHADER



The Advanced Gem Shader package contains a large library of shader to implement gemstones and especially gemstones with unusual properties like inclusions and volumetric interior effects. This allows the creation of sculpable blocks with volumetric interior effects. The extension contains a sample scene containg VoxelGenerator game objects using Gemstone material. Such object has a marching cubes to generate the surface and a UV Writer hull generator for the volumetric interior which is visualized using gemstone complex shader provided by the Advanced Gem Shader package.

FREQUENTLY ASKED QUESTIONS:

Q: Why does a VoxelGenerator has boundaries?

The underlying data structure occupies a defined volume (root size) which is fixed in order to maintain consistency. The LimitBreaker provides the option to make the VoxelGenerator limitless.

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?

The data structure itself does not care which resolution is used. But for mobile, use Width of 4 and Cell subdivision of 4 or lower. Target depth lower than 5.

Higher width means more vertices per GameObject which increases GPU efficiency but increases CPU load. Higher subdivision count but lower width means more GPU load but less CPU demand.

Increasing the Subdivision Power of the data structure reduce lookup time but at the cost of more memory.

Q: Is it possible to implement physical interaction such as floating voxels falling down?

Physical behavior in voxel engines is a very complex part and requires ton of work. Implementing this would actually double the price for this asset. Getting this implemented would require at least one more year.

Q: I need accurate voxel geometry for scientific visualization?

If you need the engine for medical or scientific stuff, it is always recommended to use Texture to voxel converter as it is the most accurate solution. Mesh to voxel converter is not accurate enough in my opinion.

Q: How do I increase the resolution?

The resolution is dynamically and the data itself can get as fine as desired by increasing the depth value of the modifier.

When increasing the depth, you also have to increase the precision of the hull generator by increasing the Width and Cell Subdivision. Finer details will be missed if the hull generator is too coarse.

For example Marching Cubes:

Generator: Subdivision Power = 2
Modifier: Depth = 7
Hull Generator: Width = 8, Cell Subdivision = 8

Generator: Subdivision Power = 2
Modifier: Depth = 8
Hull Generator: Width = 16, Cell Subdivision = 8

Generator: Subdivision Power = 2
Modifier: Depth = 9
Hull Generator: Width = 16, Cell Subdivision = 16

High Resolution:

Generator: Subdivision Power = 2
Modifier: Depth = 10
Hull Generator: Width = 32, Cell Subdivision = 16

Whenever the depth increases, you also have to double either width or cell subdivision!

If the hull generator is too fine, the result is a blocky appearance.

AUTHOR NOTES:

This voxel engine uses no external stuff which is the connected with Unity3D and therefore has no “Bridges” or other APIs as it is common in other voxel engines. Also this voxel engine was developed with full control over the engine itself in mind so every functionality can be modified. The performance is only possible through the inclusion of the Unity Job system and Burst compilation system. Therefore it is highly recommended to have those installed.

I am glad that I was allowed to work with voxels as part of my master thesis. For all those who have to write one, always choose a topic you are really interested in.

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/