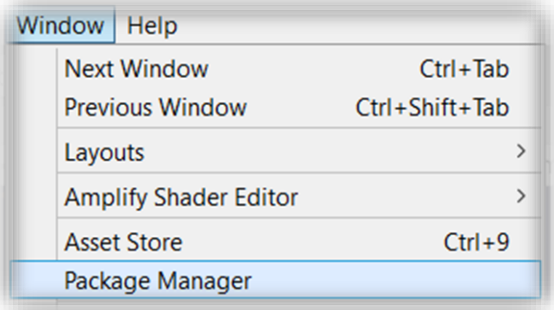
Thanks!

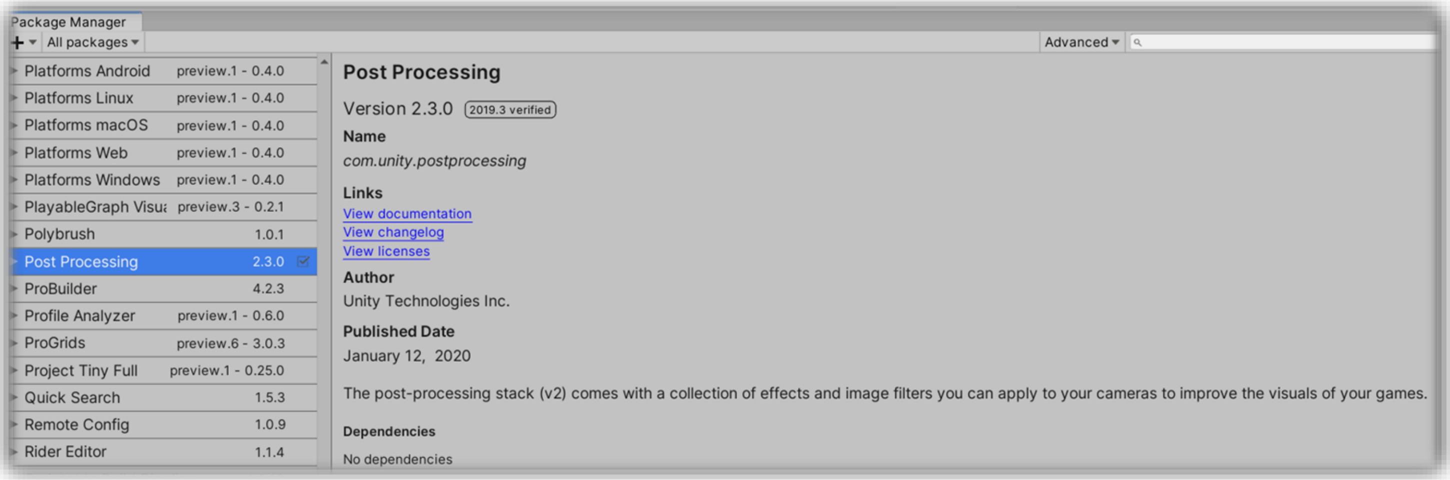
Congratulations and thanks for acquiring **Cyber Effects – Voxels**!

Check for other awesome assets on my friend's page bit.ly/DeveloperPage-Elvis and know more about the developers behind the development of this visual effects series: twitter.com/ivangarciafilho and https://twitter.com/elvismdd.

Demo scene

The demo scene requires you to download and install Unity's Post processing stack from package manager, however, even with some missing scripts, It' should run, even without all the juiciness of the post effects!





Platforms Android	preview.1 - 0.4.0
Platforms Linux	preview.1 - 0.4.0
Platforms macOS	preview.1 - 0.4.0
Platforms Web	preview.1 - 0.4.0
Platforms Windows	preview.1 - 0.4.0
PlayableGraph Visu:	preview.3 - 0.2.1
Polybrush	1.0.1
Post Processing	2.3.0 ✓
ProBuilder	4.2.3
Profile Analyzer	preview.1 - 0.6.0
ProGrids	preview.6 - 3.0.3
Project Tiny Full	preview.1 - 0.25.0
Quick Search	1.5.3
Remote Config	1.0.9
Rider Editor	1.1.4

Post Processing

Version 2.3.0 (2019.3 verified)

Name
com.unity.postprocessing

Links
View documentation
View changelog
View licenses

Author
Unity Technologies Inc.

Published Date
January 12, 2020

The post-processing stack (v2) comes with a collection of effects and image filters you can apply to your cameras to improve the visuals of your games.

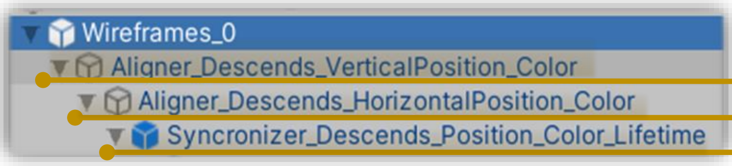
Dependencies
No dependencies

Structure Prefix and Suffix

Aligners mainly spread the particles aligned, and also descend their color downwards, randomizing the brightness of sub-emitters, and the **Synchronizer** also descends it’s **lifetime** downwards the **Renders**, designed to hold most of the visual effects composed animations (**fade-in**, **loop** and **fade-out**), by inputting **animation curves** into each parameter of the spawned particles modules.

Prefix: Aligner_

The "**Vertical aligner**" (the aligner in the hierarchy that descends vertical Position) is just an emitter without any graphical element (disabled render module), which spreads the "**Horizontal aligners**" (the other aligner that descends horizontal position) sub-emitter **"evenly"** (mostly) across the Y axis, using a **Speed** value to give it an Impression of building up from bottom to top the whole column.

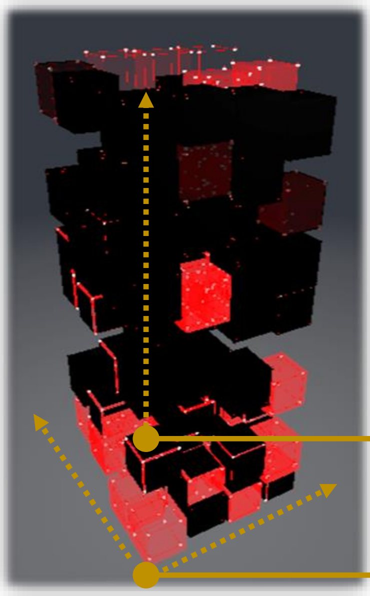


```
▼ Wireframes_0
  ▼ Aligner_Descends_VerticalPosition_Color
    ▼ Aligner_Descends_HorizontalPosition_Color
      ▼ Syncronizer_Descends_Position_Color_Lifetime
```

Aligns Vertically all rows of voxels and descends random brightness to all voxels sharing same row.

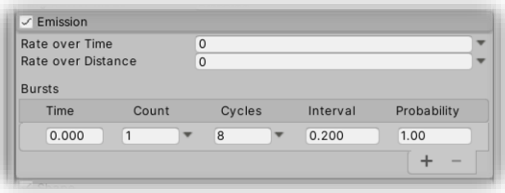
Aligns horizontally all voxels and descends random brightness to all spawned voxels.

Descends lifetime and random brightness to all spawned voxel pieces.



Since the **shape** module's **speed** parameter is something "magical", and the bursts occurs every **0.2 seconds**, within the particle duration (**1.6 seconds**):

- If You want to increase the height of the column and/or spacing between rows of voxels, you might probably try to increase the edge **radius** and **speed** parameter on some "magical" non-linear combination, and use the **position** parameter just to adjust the pivoting of the final graphical asset.

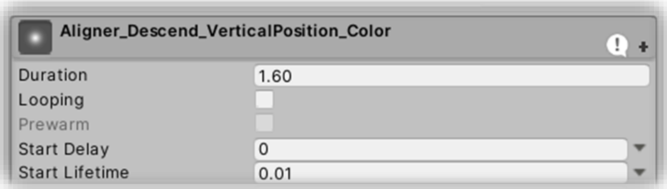


✓ Emission

Rate over Time	0
Rate over Distance	0

Bursts

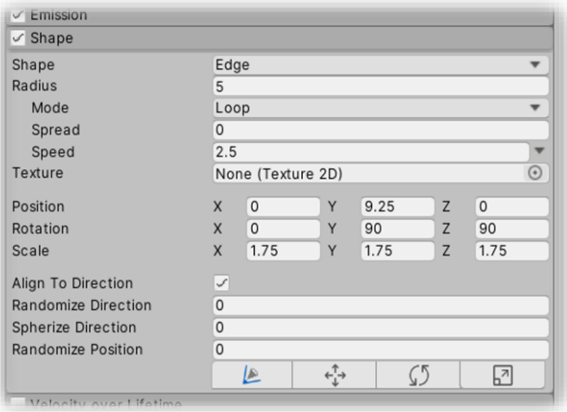
Time	Count	Cycles	Interval	Probability
0.000	1	8	0.200	1.00



Aligner_Descend_VerticalPosition_Color

Duration	1.60
Looping	☐
Prewarm	☐
Start Delay	0
Start Lifetime	0.01

- If You want to increase the "**rows**" of voxels you should also increase the **cycles** parameter (its set to **8**, so **8** rows of voxels), then increase the **duration** by the result of **interval * cycles** (don't even try to reduce the interval below **0.2** seconds, it's a magical number).

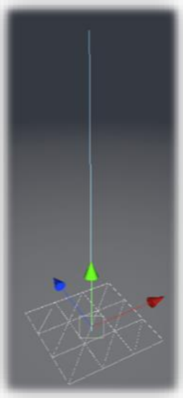


✓ Emission

✓ Shape

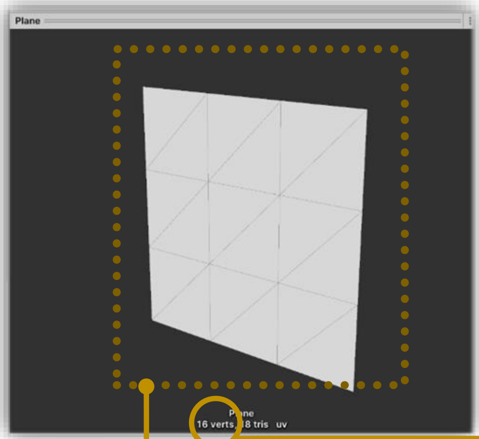
Shape	Edge					
Radius	5					
Mode	Loop					
Spread	0					
Speed	2.5					
Texture	None (Texture 2D)					
Position	X	0	Y	9.25	Z	0
Rotation	X	0	Y	90	Z	90
Scale	X	1.75	Y	1.75	Z	1.75
Align To Direction	✓					
Randomize Direction	0					
Spherize Direction	0					
Randomize Position	0					

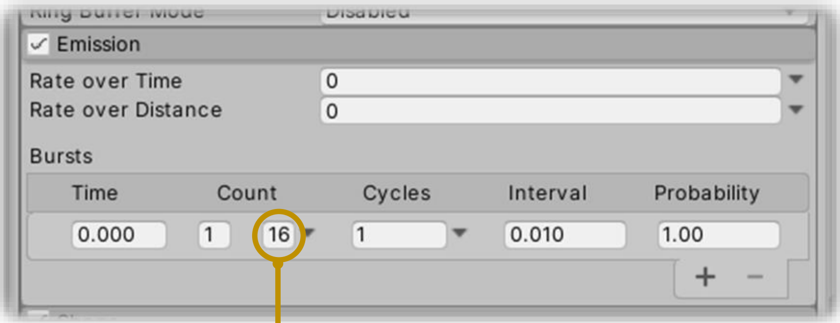
Velocity over Lifetime



The "**Horizontal aligner**" uses a **mesh** (a plane) as an input for **Shape** module's **shape** parameter to spread the voxels at precise distancing across the X and Z axis.

To work properly with the given mesh, **shape** module's **type** parameter must be set to **vertex,** the **mode** parameter set to **Loop** and **never** allow the **count** of particles spawned through the **emission** module **exceeds** the same number of vertices on input mesh.



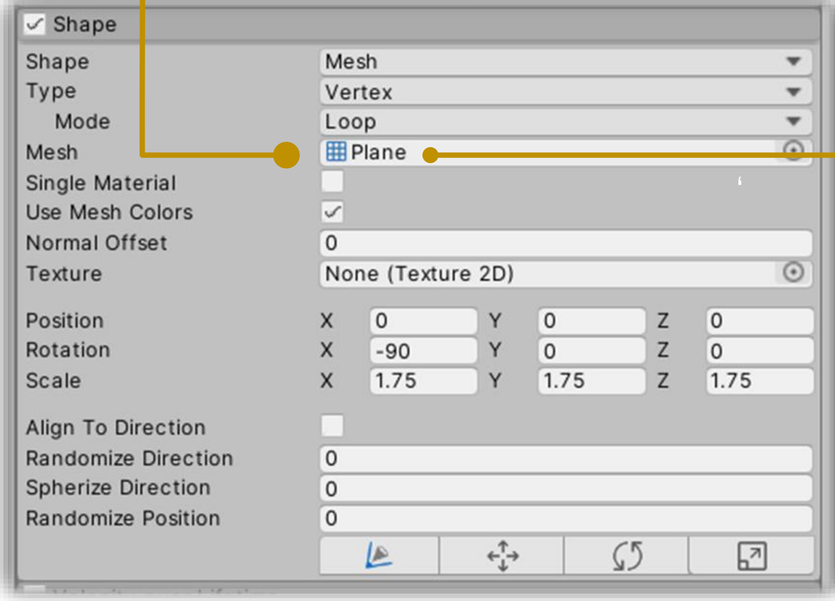


Emission

Rate over Time	0
Rate over Distance	0

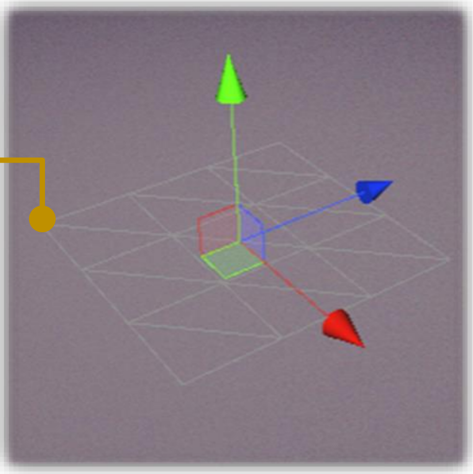
Bursts

Time	Count	Cycles	Interval	Probability
0.000	1 16	1	0.010	1.00



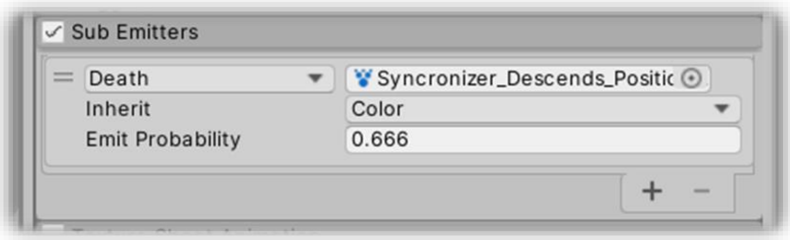
Shape

Shape	Mesh				
Type	Vertex				
Mode	Loop				
Mesh	Plane				
Single Material					
Use Mesh Colors	✓				
Normal Offset	0				
Texture	None (Texture 2D)				
Position	X 0	Y 0	Z 0		
Rotation	X -90	Y 0	Z 0		
Scale	X 1.75	Y 1.75	Z 1.75		
Align To Direction					
Randomize Direction	0				
Spherize Direction	0				
Randomize Position	0				

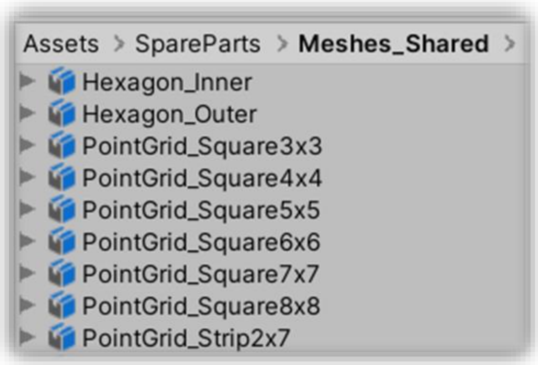


Since You can generate any grid using a mesh (on your 3D modeling tool), the **emission** module to spread the particles across all vertices, and use the **Scale** parameter of the **shape** module to adjust the spacing between every spawned voxel:

- If you want to increase the number of voxels or generate a non-uniform / non-squared formation, it is just a matter of generating a new mesh on your 3D modeling tool, then, set the **Count** on **emission** module to not exceed the amount of vertices on your mesh, and adjust the spacing between the voxels using the Scale parameter of the Shape module.
- If you want "holes" or some spots without a voxel, you should modify the **Sub-emitter** module's **Emit probability** parameter (by default it's set to **0.666**), allowing some of the aligner's particles to not spawn a voxel and then generate those chaotic / irregular shapes.
- Reducing **Emit probability** on the **Vertical aligner**, removes an entire row of voxels while lowering the **Emit probability** of every **Synchronizer's** sub-emitters, generates voxels with missing parts (**_vertex**, **_edges**, **_faces** or **_cubes**).


Sub Emitters
Death Syncronizer_Descends_Positic
Inherit Color
Emit Probability 0.666

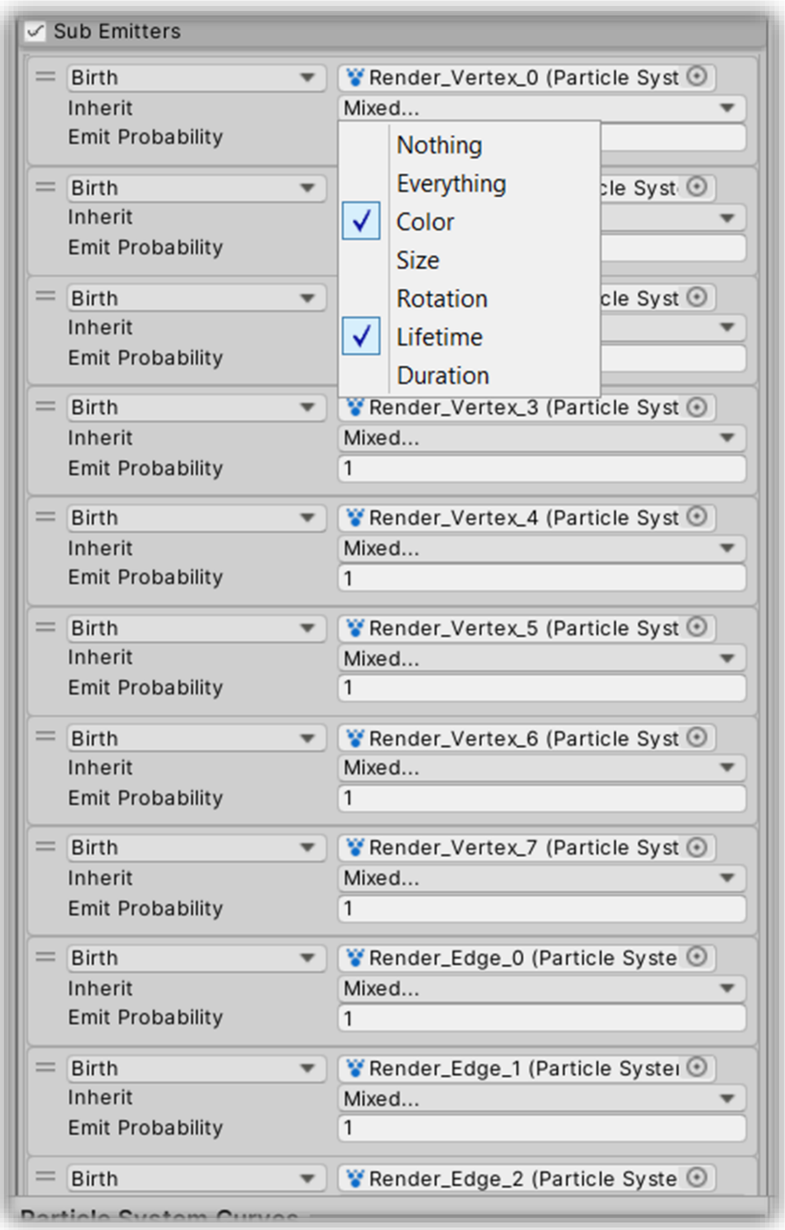
- This package comes with additional grids to spawn different particle formations. However, I strongly encourage you to generate your own horizontal grids.


Assets > SpareParts > Meshes_Shared >
Hexagon_Inner
Hexagon_Outer
PointGrid_Square3x3
PointGrid_Square4x4
PointGrid_Square5x5
PointGrid_Square6x6
PointGrid_Square7x7
PointGrid_Square8x8
PointGrid_Strip2x7

Prefix: Synchronizer_

Synchronizer mainly descends to its sub-emitters its current particle position (even if it is not a parameter shown in the **inherit** dropdown flags of the **sub-emitter** module), **Color** to generate a the voxel with some random brightness and **Lifetime** to maintain every **Render_** sub-emitter in sync with other siblings.

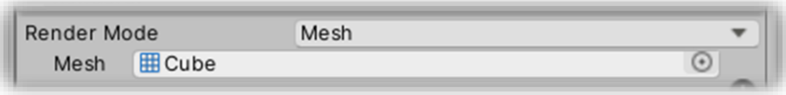
The **lifetime** is important to handle the whole child animation length, so, if you want to increase or decrease the voxel animation, you should reduce or increase the **synchronizer's lifetime**.


✓ Sub Emitters

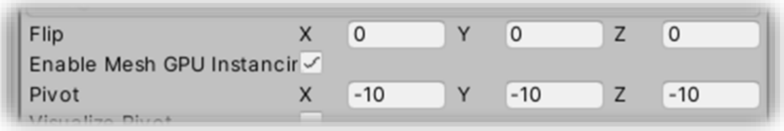
Birth	Render_Vertex_0 (Particle Syst
Inherit	Mixed...
Emit Probability	
Birth	cle Syst
Inherit	
Emit Probability	
Birth	cle Syst
Inherit	
Emit Probability	
Birth	Render_Vertex_3 (Particle Syst
Inherit	Mixed...
Emit Probability	1
Birth	Render_Vertex_4 (Particle Syst
Inherit	Mixed...
Emit Probability	1
Birth	Render_Vertex_5 (Particle Syst
Inherit	Mixed...
Emit Probability	1
Birth	Render_Vertex_6 (Particle Syst
Inherit	Mixed...
Emit Probability	1
Birth	Render_Vertex_7 (Particle Syst
Inherit	Mixed...
Emit Probability	1
Birth	Render_Edge_0 (Particle Syste
Inherit	Mixed...
Emit Probability	1
Birth	Render_Edge_1 (Particle Syste
Inherit	Mixed...
Emit Probability	1
Birth	Render_Edge_2 (Particle Syste

Prefix: Render_

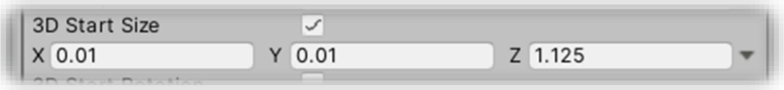
Renders are a group of **Synchronizer's** sub-emitters, emitting the mesh of **cubes** by switching the **render mode** to **Mesh** on their **Renderer** module and selecting a Unity's default **cube** primitive.



Render Mode: Mesh
Mesh: Cube



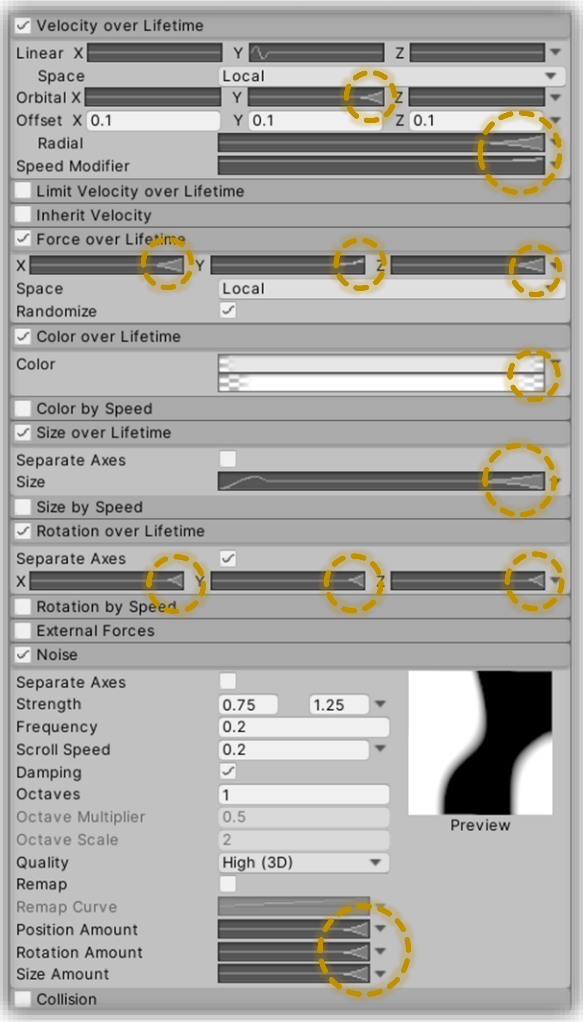
Flip: X 0, Y 0, Z 0
Enable Mesh GPU Instancir ✓
Pivot: X -10, Y -10, Z -10



3D Start Size ✓
X 0.01, Y 0.01, Z 1.125

Each with different 3D scales on their **main** module's **3D start size** and different pivot offsets assigned through **Renderer** module's **pivot** parameter to compose together the shape of a "wireframe outlined cube" and most of its synced animated parameters contributes to the resulting visual part of the effect.

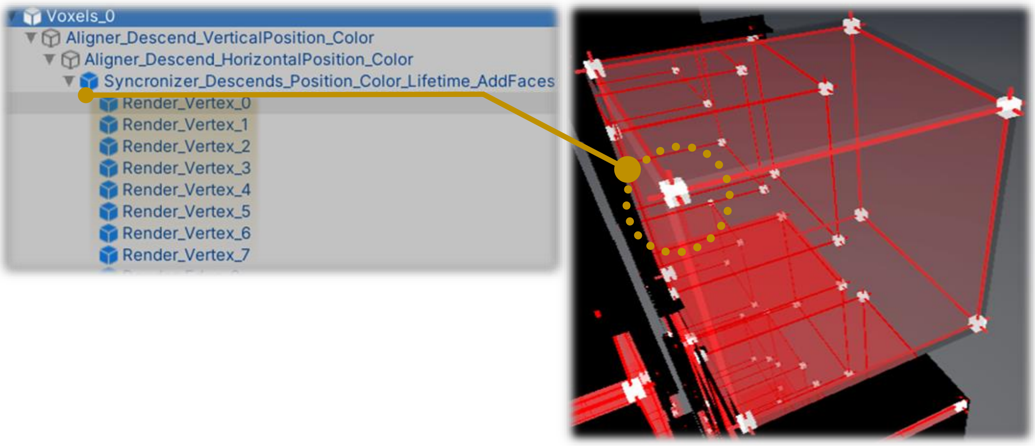
Since **Render_** sub-emitters compose the **voxel** and **wireframe**, not only their lifetime should mostly be the same, but almost every animation curve that could "move" or scale them must be the same (scale because of the pivoting technique used), otherwise you might probably "break" the aspect of a cube (exactly what happens in the fade-out animation that vanishes the voxels by breaking the sync of animated parameters, randomizing them close the **0.75** of the particles normalized lifetime).



✓ Velocity over Lifetime
Linear X, Y, Z
Space: Local
Orbital X, Y, Z
Offset: X 0.1, Y 0.1, Z 0.1
Radial
Speed Modifier
Limit Velocity over Lifetime
Inherit Velocity
✓ Force over Lifetime
X, Y, Z
Space: Local
Randomize ✓
✓ Color over Lifetime
Color
Color by Speed
✓ Size over Lifetime
Separate Axes
Size
Size by Speed
✓ Rotation over Lifetime
Separate Axes ✓
X, Y, Z
Rotation by Speed
External Forces
✓ Noise
Separate Axes
Strength: 0.75, 1.25
Frequency: 0.2
Scroll Speed: 0.2
Damping ✓
Octaves: 1
Octave Multiplier: 0.5
Octave Scale: 2
Quality: High (3D)
Remap
Remap Curve
Position Amount
Rotation Amount
Size Amount
Collision

Suffix: _Vertex

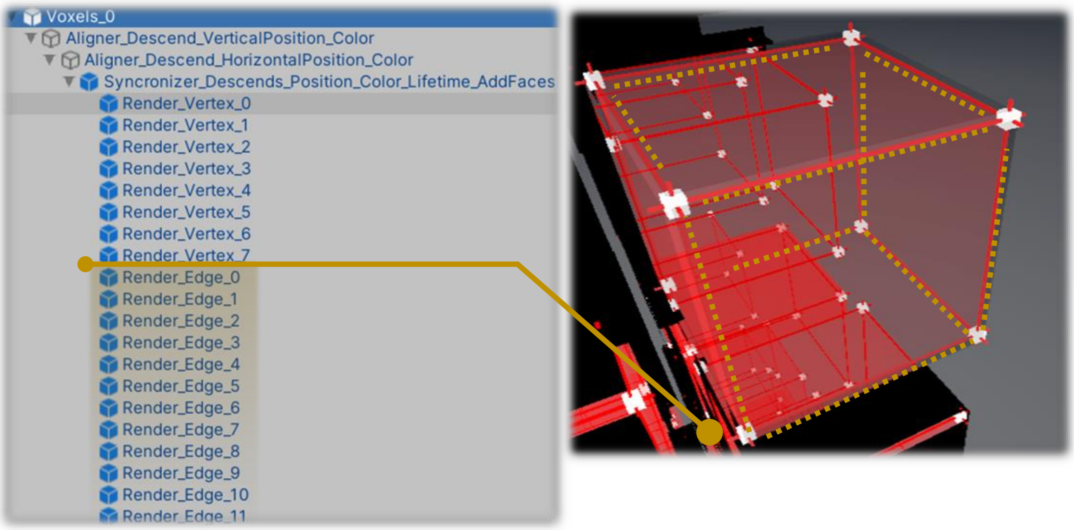
Those **Renders**, with the suffix **Vertex** draws the “white dots”, mimicking a vertex on every corner of the cube.



```
Voxels_0
  ▼ Aligner_Descend_VerticalPosition_Color
    ▼ Aligner_Descend_HorizontalPosition_Color
      ▼ Syncronizer_Descends_Position_Color_Lifetime_AddFaces
          Render_Vertex_0
          Render_Vertex_1
          Render_Vertex_2
          Render_Vertex_3
          Render_Vertex_4
          Render_Vertex_5
          Render_Vertex_6
          Render_Vertex_7
```

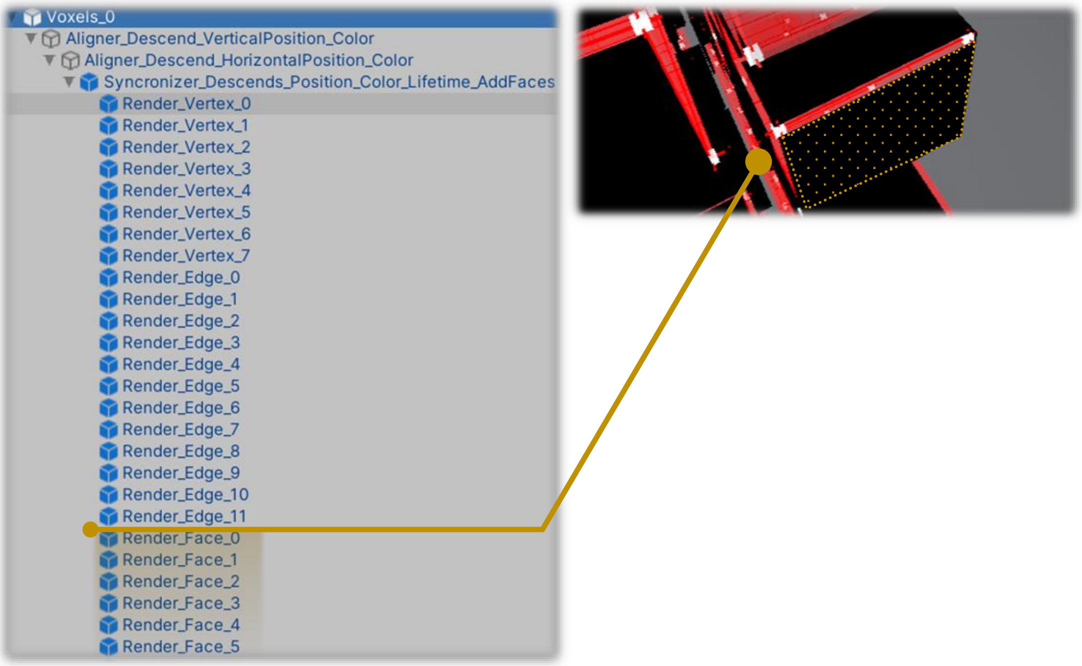
Suffix: _Edges

Those **Renders**, with the suffix **Edge** draws the “red wires”, mimicking the cubes edges like an x-ray outline.



```
Voxels_0
▼ Aligner_Descend_VerticalPosition_Color
  ▼ Aligner_Descend_HorizontalPosition_Color
    ▼ Syncronizer_Descends_Position_Color_Lifetime_AddFaces
        Render_Vertex_0
        Render_Vertex_1
        Render_Vertex_2
        Render_Vertex_3
        Render_Vertex_4
        Render_Vertex_5
        Render_Vertex_6
        Render_Vertex_7
        Render_Edge_0
        Render_Edge_1
        Render_Edge_2
        Render_Edge_3
        Render_Edge_4
        Render_Edge_5
        Render_Edge_6
        Render_Edge_7
        Render_Edge_8
        Render_Edge_9
        Render_Edge_10
        Render_Edge_11
```

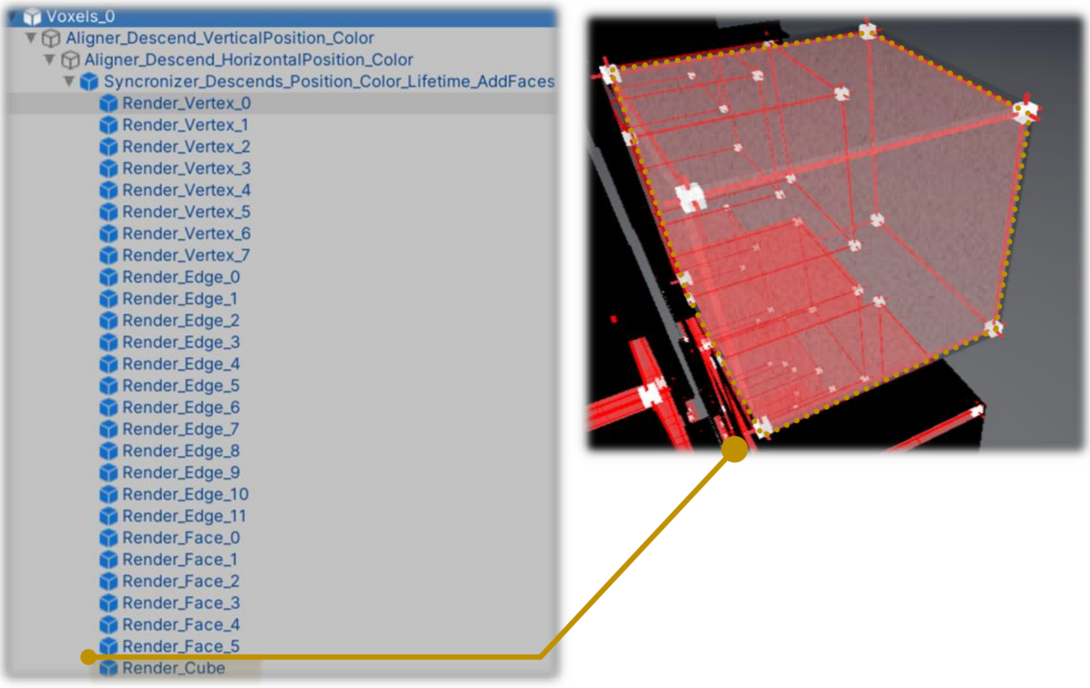
Suffix: _Face



```
Voxels_0
▼ Aligner_Descend_VerticalPosition_Color
  ▼ Aligner_Descend_HorizontalPosition_Color
    ▼ Syncronizer_Descends_Position_Color_Lifetime_AddFaces
        Render_Vertex_0
        Render_Vertex_1
        Render_Vertex_2
        Render_Vertex_3
        Render_Vertex_4
        Render_Vertex_5
        Render_Vertex_6
        Render_Vertex_7
        Render_Edge_0
        Render_Edge_1
        Render_Edge_2
        Render_Edge_3
        Render_Edge_4
        Render_Edge_5
        Render_Edge_6
        Render_Edge_7
        Render_Edge_8
        Render_Edge_9
        Render_Edge_10
        Render_Edge_11
        Render_Face_0
        Render_Face_1
        Render_Face_2
        Render_Face_3
        Render_Face_4
        Render_Face_5
```

Those **Renders**, with the suffix **Face** draws the "dark plates", mimicking the cubes faces.

Suffix: _Cube



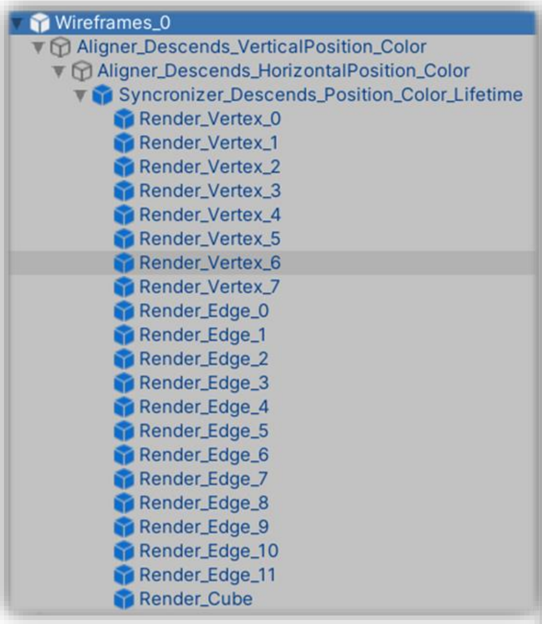
```
▼ Voxels_0
  ▼ Aligner_Descend_VerticalPosition_Color
    ▼ Aligner_Descend_HorizontalPosition_Color
      ▼ Syncronizer_Descends_Position_Color_Lifetime_AddFaces
          Render_Vertex_0
          Render_Vertex_1
          Render_Vertex_2
          Render_Vertex_3
          Render_Vertex_4
          Render_Vertex_5
          Render_Vertex_6
          Render_Vertex_7
          Render_Edge_0
          Render_Edge_1
          Render_Edge_2
          Render_Edge_3
          Render_Edge_4
          Render_Edge_5
          Render_Edge_6
          Render_Edge_7
          Render_Edge_8
          Render_Edge_9
          Render_Edge_10
          Render_Edge_11
          Render_Face_0
          Render_Face_1
          Render_Face_2
          Render_Face_3
          Render_Face_4
          Render_Face_5
          Render_Cube
```

The **Render**, with the suffix **Cube** draws the "inner red volume" used for "glows" and filling the wireframes while without the dark plating.

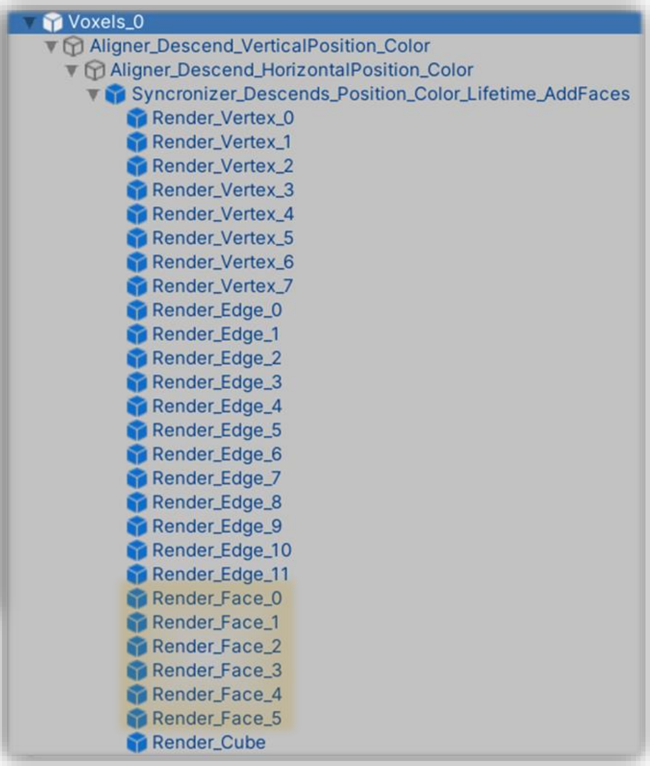
Originals and Variants

Originals and variants are mostly the same, however, I am trying to keep "variants" like a version of the originals with the smallest amount of changes as possible.

This pack comes with 2 core visual effects as the basis of every variant, **Voxels** are **Wireframes** with dark faces surrounding all the 6 sides of the cube surface by adding **6** additional **Render_** sub-emitters to its **Synchronizer_** sub-emitter.



```
▼ Wireframes_0
  ▼ Aligner_Descends_VerticalPosition_Color
    ▼ Aligner_Descends_HorizontalPosition_Color
      ▼ Syncronizer_Descends_Position_Color_Lifetime
          Render_Vertex_0
          Render_Vertex_1
          Render_Vertex_2
          Render_Vertex_3
          Render_Vertex_4
          Render_Vertex_5
          Render_Vertex_6
          Render_Vertex_7
          Render_Edge_0
          Render_Edge_1
          Render_Edge_2
          Render_Edge_3
          Render_Edge_4
          Render_Edge_5
          Render_Edge_6
          Render_Edge_7
          Render_Edge_8
          Render_Edge_9
          Render_Edge_10
          Render_Edge_11
          Render_Cube
```



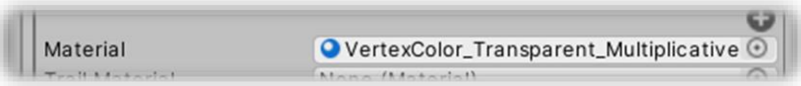
```
▼ Voxels_0
  ▼ Aligner_Descend_VerticalPosition_Color
    ▼ Aligner_Descend_HorizontalPosition_Color
      ▼ Syncronizer_Descends_Position_Color_Lifetime_AddFaces
          Render_Vertex_0
          Render_Vertex_1
          Render_Vertex_2
          Render_Vertex_3
          Render_Vertex_4
          Render_Vertex_5
          Render_Vertex_6
          Render_Vertex_7
          Render_Edge_0
          Render_Edge_1
          Render_Edge_2
          Render_Edge_3
          Render_Edge_4
          Render_Edge_5
          Render_Edge_6
          Render_Edge_7
          Render_Edge_8
          Render_Edge_9
          Render_Edge_10
          Render_Edge_11
          Render_Face_0
          Render_Face_1
          Render_Face_2
          Render_Face_3
          Render_Face_4
          Render_Face_5
          Render_Cube
```

Variants Suffixes

Loop suffix are variants with a "fade-in" animation that tends to end between **0.15** to **0.25** and looping infinitely by modifying every **Render** particle's **ring buffer mode** parameter set to **Loop** on their **main module** and removed all the least keys of every animated parameter that would ruin them seamlessly loop between **0.25** to **1** of their normalized lifetime (meaning that I removed the "fade-out" animation of every particle).

While variants with **Recolor_** mostly tend to be small modifications towards the sub-emitter's **starting color** on main module or **color over lifetime** module.

_Multiplicative and **__MultiplicativeX2** are variants that mostly change only the sub-emitter **Render_Cube** particle's **material**.



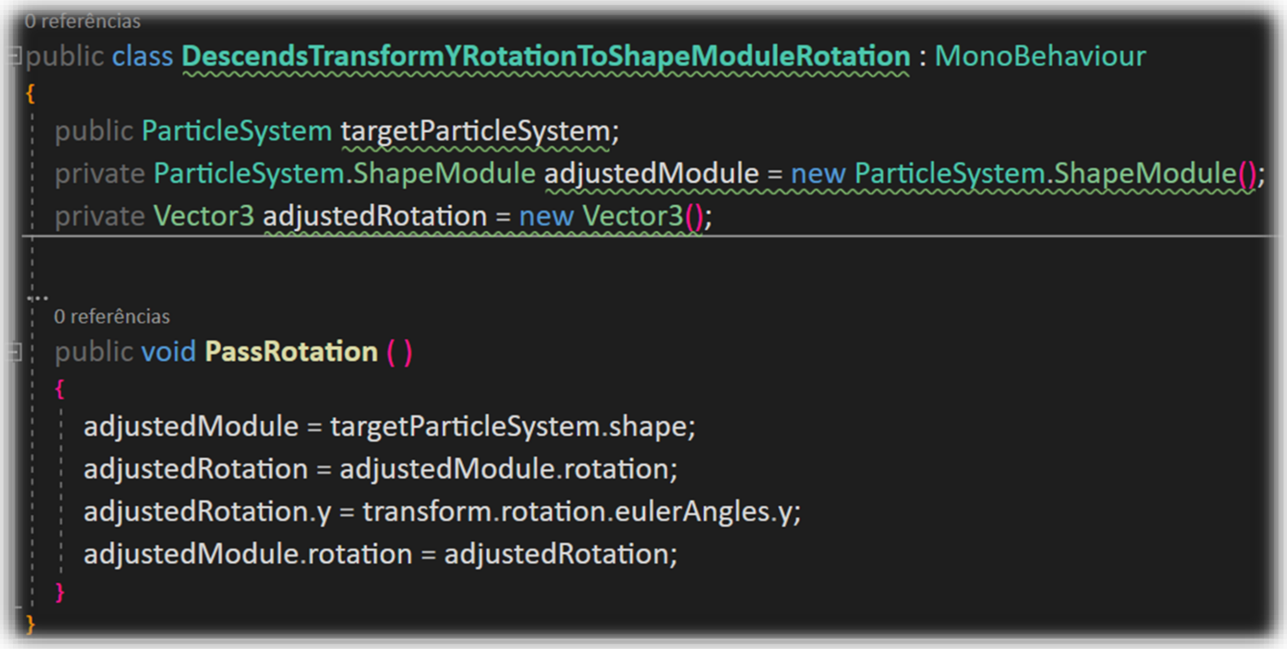
Material VertexColor_Transparent_Multiplicative

Rotation and Scripting

There is an issue (not yet solved), that is the "rotation of the particle system".

I strongly encourage you to not rotate It while playing or even try play it with any rotation along any of its axis. You should "play" with **ZERO** rotation on all of its axis, then, "pause", rotate the way you want and then "unfreeze" the particle system to continue the simulation without any visual artifacts.

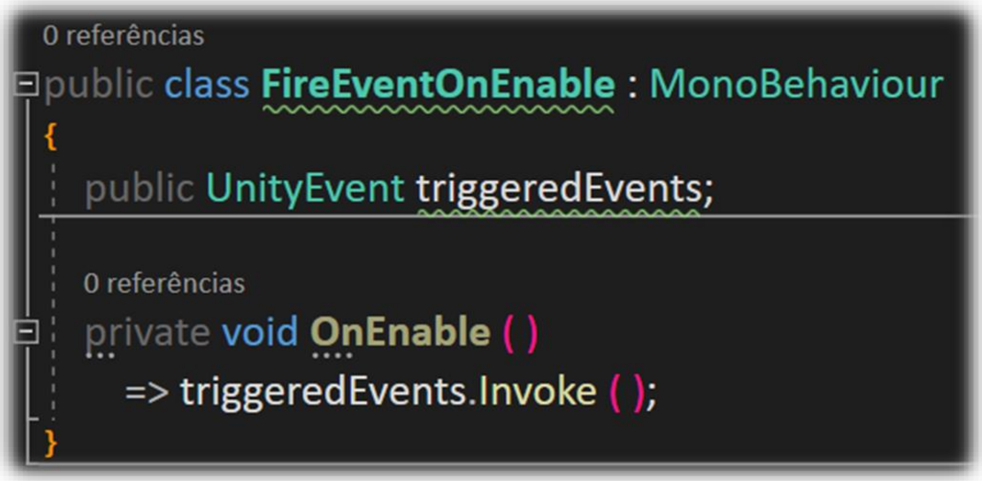
While I do not find the solution for this issue that uses **ZERO** codding, I've set an script on every prefab to pass the world rotation of the prefab's root world position down to the **horizontal aligner's** rotation (and only the Y Euler rotation);



```
0 referências
public class DescendsTransformYRotationToShapeModuleRotation : MonoBehaviour
{
    public ParticleSystem targetParticleSystem;
    private ParticleSystem.ShapeModule adjustedModule = new ParticleSystem.ShapeModule();
    private Vector3 adjustedRotation = new Vector3();


    0 referências
    public void PassRotation ( )
    {
        adjustedModule = targetParticleSystem.shape;
        adjustedRotation = adjustedModule.rotation;
        adjustedRotation.y = transform.rotation.eulerAngles.y;
        adjustedModule.rotation = adjustedRotation;
    }
}
```

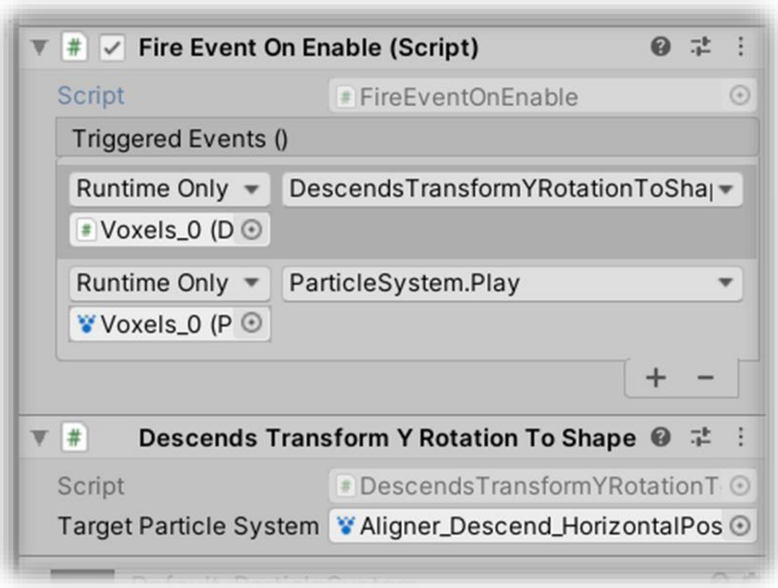
And another script just to fire events on enable.



```
0 referências
public class FireEventOnEnable : MonoBehaviour
{
    public UnityEvent triggeredEvents;

    0 referências
    private void OnEnable ( )
        => triggeredEvents.Invoke ( );
}
```

This way, I could expose the logic through the inspector and let you also tie your own logic without overriding the scripts.

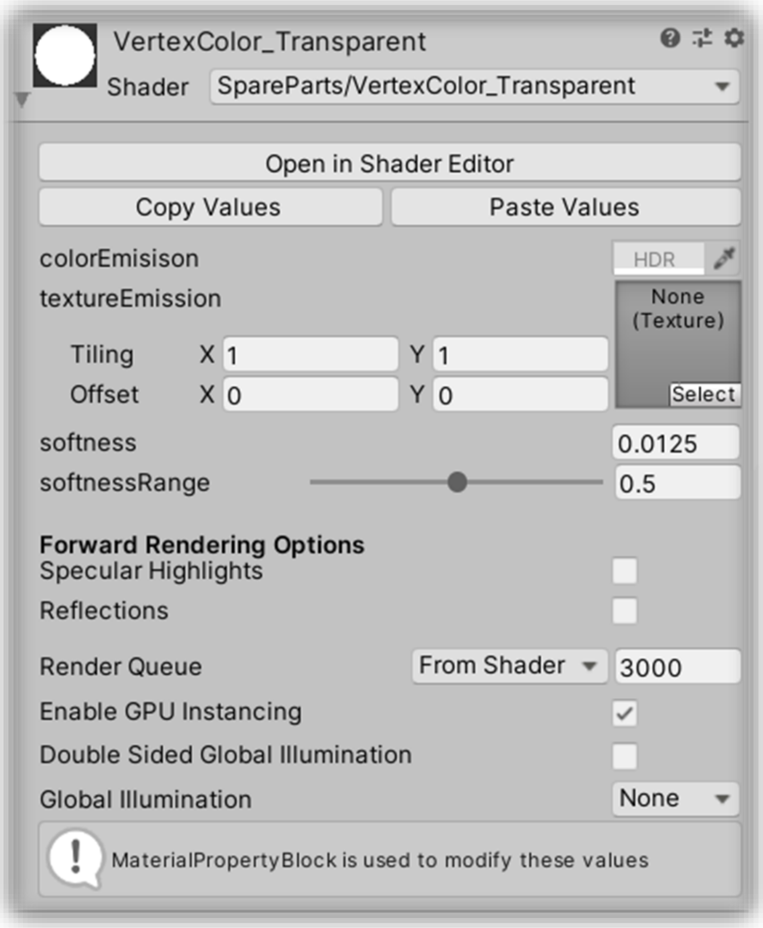

Fire Event On Enable (Script)
Script FireEventOnEnable
Triggered Events ()
Runtime Only DescendsTransformYRotationToSha|
Voxels_0 (D
Runtime Only ParticleSystem.Play
Voxels_0 (P
Descends Transform Y Rotation To Shape
Script DescendsTransformYRotationT
Target Particle System Aligner_Descend_HorizontalPos

As you can see above, every prefab will first apply the rotation and after that play the particle system,

Shaders

Simple Unlit shaders that I added some brightness control through particle system's custom vertex streams, this way, I could use 1 single material for everything and then manage to make the black plating emissive or not, so the bloom post effect would capture their color brightness exceeding or not the threshold.

The **VertexColor_Transparent** is an alpha blended unlit shader and the only difference to other variants (multiplicative or multiplicativex2) uses different blending techniques.



VertexColor_Transparent

Shader SpareParts/VertexColor_Transparent

colorEmisison		HDR
textureEmission		None (Texture) Select
Tiling	X 1	Y 1
Offset	X 0	Y 0
softness		0.0125
softnessRange		0.5

Forward Rendering Options

Specular Highlights	☐
Reflections	☐
Render Queue	From Shader 3000
Enable GPU Instancing	✓
Double Sided Global Illumination	☐
Global Illumination	None

MaterialPropertyBlock is used to modify these values

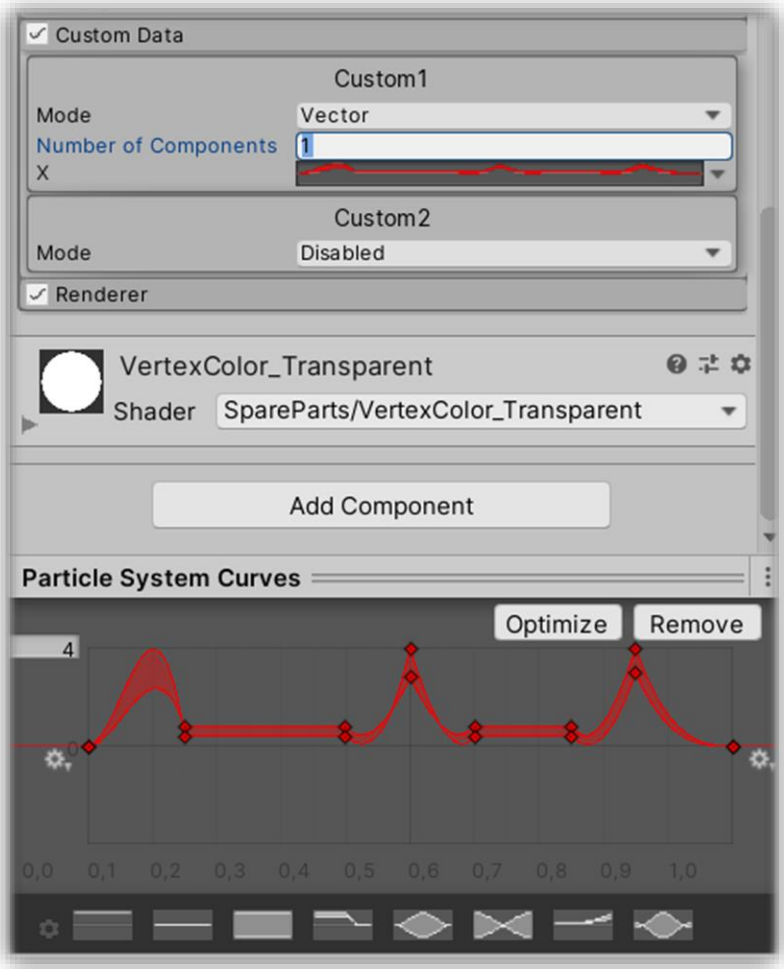
colorEmission multiplies the particle color by chosen color and the texture does the same, injecting that value straight into the emission channel.
softness and **softnessRange** smooths the intersections between particle and geometry.

Those shaders are fully compatible with **Amplify shader** and if you have your own shader, you should just read the incoming vertex **TEXTCOORD0.w** and use the input value to multiply the color that goes into the emission channel.

Custom Vertex Streams

Every **Render_** particle is sending to the shader through the **custom data** module a float parameter generated by an animation curve evaluated over its lifetime.

Allowing the particle system to handle the brightness of the emission color.



Custom Data

Custom1

Mode: Vector

Number of Components: 1

X

Custom2

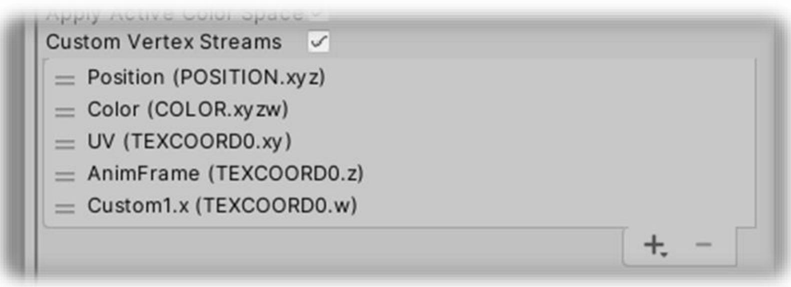
Mode: Disabled

Renderer

VertexColor_Transparent

Shader: SpareParts/VertexColor_Transparent

On **renderer** module, the **custom vertex stream** is toggled on and may contain some useless parameters being send towards the shader, just because I had to make all particles use exactly the same **TEXCOORD.W**, otherwise, my shader would not read the right parameter when multiplying the emission color.



Custom Vertex Streams ✓

Position (POSITION.xyz)

Color (COLOR.xyzw)

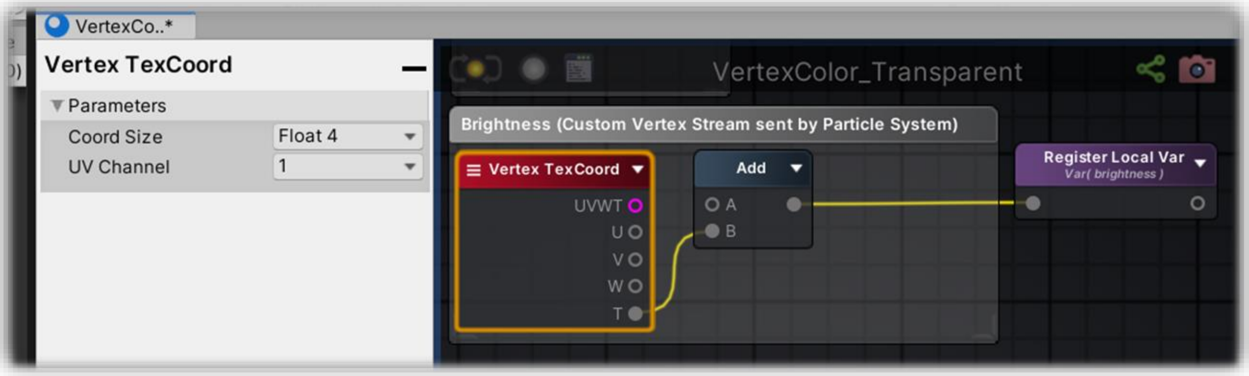
UV (TEXCOORD0.xy)

AnimFrame (TEXCOORD0.z)

Custom1.x (TEXCOORD0.w)

That’s a bit weird to explain, however, for amplify shader, the UV CHANNEL 1 is the UVCHANNNEL 0 of the vertex of the particle, they start counting from 1 instead of 0.

Then since **UVCHANNEL = TEXCOORD**, thus **TEXCOORD.W = UV.T** within amplify shader.


Vertex TexCoord

Parameters

Coord Size: Float 4

UV Channel: 1