May 20, 2020

TT Modding Resource: Particle Systems (Unity 2017.4)

FLSoz

May 20, 2020

Contents

1 Introduction

This resource is meant to serve as an introductory guide to using Unity's Particle Systems in the **specific scope** of TerraTech CustomBlocks, for use with the BlockInjector. If you want to learn more about UnityEngine.ParticleSystems in general, visit the official Unity Manual. For modders, you may find the official Script API more useful, as it provides field names.

Take care to check the version of the docs when reading, as newer versions may have changed interfaces, which means your attempts may fail. I postulate that the Unity version of most relevance is 2017.4, but have not asked for, or received confirmation as of writing this guide.

This guide assumes a working knowledge of references in regards to how the BlockInjector handles them. Knowing the basic theory behinds pointers, or even class fields will be **immensely** useful.

2 First Steps

Unfortunately, you are unable to create new ParticleSystems purely from scratch using the JSON injector. While you are perfectly able to create new ParticleSystem objects, there are some fields which you will need a prior reference for (I will elaborate later).

Unity, as a game engine, runs on entities called "GameObject"s. Everything (that you care about, anyway) is attached to a GameObject, either directly, or through some hierarchy. As an item of class "Component", ParticleSystem, and ParticleSystemRenderer must both be directly attached to a GameObject in order for them to work.

Through BlockInjector JSON files, you will only be able to create ParticleSystems directly in a GameObject, or in a class/module that has a specific field that needs a ParticleSystem type. The same restrictions apply to ParticleSystemRenderers.

It is potentially possible for you to directly attach multiple ParticleSystem and ParticleSystemRenderer pairs to the same GameObject. This is something I, personally, have not experimented with. Given that there is some linkage with Unity rendering ParticleSystems using the ParticleSystemRenderer attached to the same GameObject, I would not recommend it. Instituting a hierarchy is useful, both for finer control, cleanliness, and safety reasons.

I.E.: Instead of having a structure in your JSON of form: (note that the $< 1 >$ is because it's not a valid JSON otherwise, and would not be in your actual JSON)

```
{
    "GameObject|root": {
        "ParticleSystem <1>": {},
        "ParticleSystemRenderer <1>": {},
        "ParticleSystem <2>": {},
        "ParticleSystemRenderer <2>": {}
    }
}
```

Use a structure of the form below instead

```
{
    "GameObject|root": {
        "GameObject|particles1": {
            "UnityEngine.Transform": {},
            "ParticleSystem": {},
            "ParticleSystemRenderer": {}
        },
        "GameObject|particles2": {
            "UnityEngine.Transform": {},
            "ParticleSystem": {},
            "ParticleSystemRenderer": {}
        }
    }
}
```

Note the introduction of the Transform components. With individualized Transform components (they don't share one), you can independently control the location and scale of the two ParticleSystems.

3 Creating Your First ParticleSystem

If you want to create a new ParticleSystem, you should create one using "UnityEngine.ParticleSystem". If you are referencing an extant ParticleSystem, you have several options, and how to proceed depends on **how** you're referencing it.

3.1 Case 1: No referencing

The ParticleSystem is cleanly defined in a GameObject Hierarchy (relatively rare)

```
{
    "GameObject|root": {
        "GameObject|particles1": {
            "UnityEngine.Transform": {},
            "ParticleSystem": {},
            "ParticleSystemRenderer": {}
        }
    }
}
```

No special things need to be taken care of here. Modify parameters directly in the hierarchy as appropriate. (to be described later)
Do be aware that any extant game code that hooks to it may force it to play at unwelcome times (rare - theoretically possible, haven't seen).
The closest case to that I've seen so far is MuzzleFlash code, which will play any ParticleSystem directly under "GameObject|_muzzleFlash". (can be different name, mostly is as described)

3.2 Case 2: Referencing

The ParticleSystem is on another block, which is cleanly defined in a GameObjectHierarchy (pretty common)

```
{
    "GameObject|root": {
        "Reference|<BlockPrefabName>/<GameObjectHierarchy>/<ParentGameObject>": {
            "UnityEngine.Transform": {},
            "ParticleSystem": {},
            "ParticleSystemRenderer": {}
        }
    }
}
```

Similar to Case 1, no special things need to be taken care of, and you can directly modify parameters. Unlike Case 1, you will not need to worry about extant triggers being there, because you copied it from a different prefab. The exception lies in positional things, like "GameObject|_muzzleFlash", which will play any ParticleSystem directly under that GameObject.

3.3 Case 3: Self-Referencing

The ParticleSystem is somewhere else on the block you're building. This scenario actually comes in several different flavors:

```
{
    "GameObject|root": {
        "GameObject|Hierarchy": {
            "GameObject|particles1": {
                "UnityEngine.Transform": {},
                "ParticleSystem": {},
                "ParticleSystemRenderer": {}
            }
        },
        "Duplicate|/root/Hierarchy/particles1": {
            "UnityEngine.Transform": {},
            "ParticleSystem": {},
            "ParticleSystemRenderer": {}
        }
    }
}
```

Most of the time, the same caveats as Cases 1 & 2 apply, and that's that. Some things to note:

1. Duplicating/Referencing a Parent of the Parent GameObject of a ParticleSystem faithfully captures all information (and duplicates - creating a separate instance. No shallow copies here)
2. If duplicating the direct Parent of a ParticleSystem, which you have referenced as in Case 4, you will have to set the Transform component of the copy as well (why will be explained in Case 4)

```
{
    "GameObject|root": {
        "GameObject|Hierarchy": {
            "GameObject|particles1": {
                "UnityEngine.Transform": {},
                "ParticleSystem": {},
                "ParticleSystemRenderer": {}
            }
        },
        "Duplicate|/root/Hierarchy": {
            "GameObject|particles1": {
                "UnityEngine.Transform": {},
                "ParticleSystem": {},
                "ParticleSystemRenderer": {}
            }
        }
    }
}
```

```
{
    "GameObject|root": {
        "GameObject|Hierarchy": {
            "GameObject|particles1": {
                "UnityEngine.Transform": {},
                "ParticleSystem": {},
                "ParticleSystemRenderer": {}
            }
            "Duplicate|particles1": {
                "UnityEngine.Transform": {},
                "ParticleSystem": {},
                "ParticleSystemRenderer": {}
            }
        }
    }
}
```

3.4 Case 4: Unclean Referencing

You are referencing a ParticleSystem on a block prefab, which is inaccessible outside of special circumstances (Plasma Beam effects, etc). The block prefab referenced may or may not be different than the one you're working on

```
{
    "GameObject|root": {
        "Reference|HE_PlasmaTeeth_311/_barrel/_beam/BeamWeapon.m_BeamParticlesPrefab/": {
            "UnityEngine.Transform": {},
            "ParticleSystem": {},
            "ParticleSystemRenderer": {}
        }
    }
}
```

```
{
    "GameObject|root": {
        "GameObject|UselessName": {
            "UnityEngine.Transform": {},
            "Reference|HE_PlasmaTeeth_311/_barrel/_beam/BeamWeapon.m_BeamParticlesPrefab/
                        ParticleSystem.": {},
            "Reference|HE_PlasmaTeeth_311/_barrel/_beam/BeamWeapon.m_BeamParticlesPrefab/
                        ParticleSystemRenderer.": {}
        }
    }
}
```

Sometimes, there's just no getting around it. You have to get your hands dirty and reference something ugly. There are two ways to access ParticleSystems this way, as displayed above.

Both will do the same thing (except Transform info - haven't extensively tested if Transforms do identical things between the two, I mostly use the top version)

Note how I have titled the parent GameObject "GameObject|UselessName". That's an artifact of Unity + the BlockInjector. See, while the ParticleSystem we're trying to access isn't accessible through a direct GameObject hierarchy from the root GameObject of the prefab block, it's still attached to a GameObject. As you may remember from your Transforms knowledge, "Reference|HE_PlasmaTeeth_311/_barrel/_beam/BeamWeapon.m_BeamParticlesPrefab/" forces the BlockInjector to find the transform object of the m_BeamParticlesPrefab Component of the BeamWeapon class.

A Unity quirk happens to be that GameObjects are identified by metadata located in their Transforms. Thus, "Reference|HE_PlasmaTeeth_311/_barrel/_beam/BeamWeapon.m_BeamParticlesPrefab/" fetches the GameObject that is the direct parent to the desired particles (and which is why the Top version is the more refined way of doing things, which I prefer). Now, getting back on track, UselessName is called UselessName, because the BlockInjector copies over all data when copying the ParticleSystem in the bottom example, including the transform, which overrides the default name of UselessName into what it would normally be. (Use Misc Mods exporter to find out what that is)

However, a downside of this Transform trickery to get otherwise unobtainable stuff is that the Transforms themselves, while valid, contain incorrect information. As such, you must re-specify where to stick the ParticleSystem by forcibly overwriting the default copied-over UnityEngine.Transform.

4 The ParticleSystemRenderer Class

The UnityEngine.ParticleSystemRenderer class controls how linked ParticleSystems display their components. The full API can be found on the Script API, but I've picked out the most useful ones below:

```
namespace UnityEngine;
public class ParticleSystemRenderer {
    // Control the direction that particles face.
    public ParticleSystemRenderSpace alignment;
    // How particles are drawn.
    public ParticleSystemRenderMode renderMode;

    // Set the material used for each particle
    public Material material;
    // Set the material used by the Trail module for attaching trails to particles.
    public Material trailMaterial;

    // Clamp the minimum particle size.
    public float minParticleSize;
    // Clamp the maximum particle size.
    public float maxParticleSize;

    // How much are billboard particle normals oriented towards the camera.
    // Spherizes billboard normals when set to 0,
    //    and points them towards the camera when set to 1.
    public float normalDirection;
    // Modify the pivot point used for rotating particles.
    // The units are expressed as a multiplier of the particle sizes, relative to their diameters.
    //    For example, a value of 0.5 would adjust the pivot by the particle radius,
    //    allowing particles to rotate around their edges.
    public vector3 pivot;

    // Mesh used as particle instead of billboarded texture.
    public Mesh mesh;
    // The number of meshes being used for particle rendering. (no idea what that means)
    public int meshCount;

    // Biases particle system sorting amongst other transparencies.
    // Use lower (negative) numbers to prioritize the particle system to draw closer to the front
    public float sortingFudge;
    // Sort particles within a system.
    public ParticleSystemSortMode sortMode;

    // How much are the particles stretched depending on the Camera's speed.
    // Use this to make particles become large if the viewing camera has a large speed.
    public float cameraVelocityScale;
    // How much are the particles stretched in their direction of motion.
    // Use this to make particles always be longer than they are wide.
    public float lengthScale;
    // How much are the particles stretched depending on "how fast they move".
    public float velocityScale;
}
```

The fields you'll need to edit most often are: alignment, renderMode, material, and trailMaterial

4.1 Materials

Materials are one of the fundamentally most important parts of the Particle System as a whole. Without materials, your system will render a bunch of magenta squares, or depending on settings, magenta trails.

Unfortunately, Materials are normally created in the Unity editor, or through C# scripts that take, as input, another Material or a Shader as a base. Since there is no Material() constructor, the Block Injector **CANNOT** create new Materials, and thus **CANNOT** create new Particle Systems ex nihilo.

This is the true reason that we are forced to use Reference and Duplicate to create ParticleSystem effects.

The more astute among you will have realized that this restriction applies only to ParticleSystemRenderer instances. I have not tested using referenced ParticleSystemRenderers with fully user-created ParticleSystems. That being said, I still recommend referencing ParticleSystems, as in the majority of scenarios, you will want characteristics already present somewhere, and referencing means less work on your end.

That being said, we only use Materials (in this context) in two places: material and trailMaterial. While there is an inherited sharedMaterial field, that serves as a pointer to a shared material space (shocker, I know), and changing that, will also change the material of the original prefab you're referencing (oops). Unfortunately, being limited to these two fields means we're somewhat limited in what we're able to do. But, given those other options are only practically accessible through C# scripting, JSON block modders shouldn't see much difference.
This is the interface for the Material class: (at least the parts we're able to access)

```
namespace UnityEngine;
public class Material {
    // color of Material (seems to be overriden by ParticleSystem settings, and actual texture color)
    public Color color;
    // Controls Double Sided Global Illumination. (fancy lighting stuff)
    public bool doubleSidedGI;
    // Defines how to interact with lightmaps and probes. (more fancy lighting stuff)
    public MaterialGlobalIlluminationFlags globalIlluminationFlags;

    // enables GPU instancing
    public bool enableInstancing;

    // Directly control what texture you provide
    public Texture mainTexture;
    public Vector2 mainTextureOffset;
    public Vector2 mainTextureScale;

    // The following are Shader-specific
    public int renderQueue;
    public Shader shader;
    public string[] shaderKeywords;
}
```

The main thing you'll want to edit is mainTexture, and potentially offset/scale as appropriate. While the Block Injector does not offer a way to create Materials ex nihilo, it does offer a way to create Textures out of file paths.
As you'll see on the next page, it's possible to take advantage of that functionality to swap out the material texture being used on an object, be it a tank shell, or a Particle system.

The following is a snippet of JSON code from @Potato's Knightfall Nuclear Cannon.

```
{
    "UnityEngine.MeshRenderer": {
        "material": {
            "mainTexture": "LK_tex_1.png"
        }
    }
}
```

You will notice that while he is modifying a MeshRenderer, "material" as a field is actually inherited from the "Renderer" class, which ParticleSystemRenderer also inherits. Using that same syntax will be enough for our purposes.
In the case you want fine-grain control over other aspects of the texture, and you haven't changed the actual texture source, you are able to modify the parameters in the following:

```
public class Texture {
    // controls Texture Anisotropic filtering (makes better at shallow angle, performance cost)
    // ranges from 1 to 9, 1 is no filtering, 9 is full filtering
    public int anisoLevel;

    // controls appearances
    //    Point - texture pixels get blocky up close
    //    Bilinear - texture samples averaged
    //    Trilinear - texture samples averaged and blended between mipmap levels
    public FilterMode filterMode

    // Positive bias ==> texture is extra blurry
    // Negativie bias ==> sharpens texture (at performance cost)
    // recommended always use > -0.5 (use anisotropic for better looks)
    public float mipMapBias;

    // control how texture wraps. Set for different axes using wrapModeU, wrapModeV, wrapModeW
    //    Repeat - tile the texture
    //    Clamp - clamp texture to last pixel at the edge
    //    Mirror - Tile the texture, mirroring at every integer boundary
    //    MirrorOnce - Mirrors the texture once, then clamps to edge
    public TextureWrapMode wrapMode;
    public TextureWrapMode wrapModeU;
    public TextureWrapMode wrapModeV;
    public TextureWrapMode wrapModeW;
}
```

In the case you have changed the texture, however, there is only one way to modify these settings. You must disable your ParticleSystem (which we will learn how to do later), and create a duplicate reference, in which you must re-enable it, to be able to modify those settings. Overall, given the complexity level of modifying it, it's just not worth it for most applications.

The more common scenario you will encounter is referencing another material, which you can do as follows:

```
{
    "ParticleSystemRenderer": {
        "trailMaterial": "Reference|HE_PlasmaTeeth_311/_barrel/_beam/BeamWeapon.
                          m_BeamParticlesPrefab/ParticleSystemRenderer.material."
    }
}
```

4.2 Alignment

Alignment describes how particles are aligned by the renderer. It is likely you will want the particles to face the camera in order to look good. For the vast majority of use cases, you will always be using alignment 0, or "View".

```
namespace UnityEngine;
public enum ParticleSystemRenderSpace {
    View,       // Particles face the camera plane
    World,      // Particles align with the world space
    Local,      // Particles align with their local transform
    Facing,     // Particles face the eye position
    Velocity    // Particles align to their direction of travel
}
```

4.3 Render Mode

Render mode describes how, if at all, particles are rendered. The majority of cases will have you using "Billboard". For use cases such as beam particles, you may want to consider switching to "Stretch". For applications where you're more interested in the trails the particles leave behind, you may even want to consider "None".

```
namespace UnityEngine;
public enum ParticleSystemRenderMode {
    Billboard,              // Render particles as billboards always facing the camera
    Stretch,                // Stretch particles in direction of motion. 0 velocity = no render
    HorizontalBillboard,    // Render particles as billboards always facing up along Y axis
    VerticalBillboard,      // Render particles as billboards always facing the player,
                            // but steady on the x Axis
    Mesh,                   // Render particles as meshes. Use with ParticleSystemRenderer.mesh
    None                    // do not render any particles
}
```

There is empty space here. Time to fill with gibberish to be replaced when new wisdom pops up. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Non consectetur a erat nam. Augue mauris augue neque gravida in fermentum et. Aliquam vestibulum morbi blandit cursus. Viverra ipsum nunc aliquet bibendum. Dapibus ultrices in iaculis nunc sed. Ut enim blandit volutpat maecenas volutpat blandit aliquam etiam erat. Vitae tortor condimentum lacinia quis vel. Ipsum dolor sit amet consectetur adipiscing. Venenatis a condimentum vitae sapien pellentesque habitant. Viverra justo nec ultrices dui. Viverra nibh cras pulvinar mattis. Nisl nisi scelerisque eu ultrices vitae. Malesuada fames ac turpis egestas maecenas pharetra. Sed viverra ipsum nunc aliquet bibendum enim facilisis gravida neque. Tristique senectus et netus et malesuada fames ac. Fringilla urna porttitor rhoncus dolor purus non. Elementum facilisis leo vel fringilla.

5 The ParticleSystem Class

The UnityEngine.ParticleSystem class controls the bulk of the configuration aspects. Everything from how large, to how long they last, to even the color of particles are defined in here. The full API can be found on the Script API, but I've picked out the most useful ones below:

```
namespace UnityEngine;
public class ParticleSystem {
    // The 3 most important modules
    public ParticleSystem.MainModule main;         // Triumvirate member #1
    public ParticleSystem.EmissionModule emission; // Triumvirate member #2
    public ParticleSystem.ShapeModule shape;       // Triumvirate member #3

    // handles particle colors
    public ParticleSystem.ColorBySpeedModule colorBySpeed;
    public ParticleSystem.ColorOverLifetimeModule colorOverLifetime;

    // handles particle rotations
    public ParticleSystem.RotationBySpeedModule rotationBySpeed;
    public ParticleSystem.RotationOverLifetimeModule rotationOverLifetime;

    // handles particle sizes
    public ParticleSystem.SizeBySpeedModule sizeBySpeed;
    public ParticleSystem.SizeOverLifetimeModule sizeOverLifetime;

    // velocity modules
    public ParticleSystem.InheritVelocityModule inheritVelocity;
    public ParticleSystem.VelocityOverLifetimeModule velocityOverLifetime;
    public ParticleSystem.LimitVelocityOverLifetimeModule limitVelocityOverLifetime;

    // handles different/changing particle appearance through texture sheets
    public ParticleSystem.TextureSheetAnimationModule textureSheetAnimation;

    // handles particle light emission
    public ParticleSystem.LightsModule lights;

    // handles Particle movement noise
    public ParticleSystem.NoiseModule noise;

    // Handles particle trails
    public ParticleSystem.TrailModule trails;

    // Handles how particles collide with terrain/stuff
    public ParticleSystem.CollisionModule collision;

    // random seed stuff - you don't care unless you want deterministic randomness
    public uint randomSeed;         // Override random seed used
    public bool useAutoRandomSeed;  // Controls whether the Particle System uses an
                                    // automatically-generated random number to seed
                                    // the random number generator.
}
```

There are several modules that I have ommitted from this list. Those would be the following:

1. **ExternalForcesModule** concerns itself with particles reacting to Wind. TT does not have wind.
2. **ForceOverLifetimeModule** concerns itself with particles reacting to Wind. TT does not have wind.
3. **TriggerModule** relies on C# script function calls to set itself up
4. **SubEmittersModule** relies on C# script function calls to set itself up
5. **CustomDataModule** relies on C# script function calls to set itself up

In each and every case, trying to modify them through JSON would be an exercise in futility (and frustration).
First, an important note about the ParticleSystem modules. To the less code-inclined of us, I give you the following warning:

ParticleSystem Modules should not be referenced. Ever.

To the more code-inclined, here is the reason:

The Unity ParticleSystem is actually almost wholly coded in a C++ backend.The modules you get when accessing the ParticleSystem, e.g. system.main, are actually empty structs, whose only purpose is to represent the appropriate sections in the C++ backend. Modifying these values will automatically set the appropriate things in the backend.

The BlockInjector **should** fail when you try to reference a module. As I have not tested it, I am not actually sure if it does. If it miraculously succeeds, then you had best hope it only copied over the values, and not the pointer reference to the original ParticleSystem, or you may have unintentional overwrites.
I repeat: the above is pure speculation, and rather unlikely, but it is a possibility. Instead, you should only modify the sections you need, making use of prefabs I will provide if necessary.

The point stands as thus: just do not attempt to reference ParticleSystem modules. Class parts of them, sure. The entire module? no.

5.1 The MinMaxCurve Struct

The ParticleSystem.MinMaxCurve struct is not actually a module (nor is the MinMaxGradient struct). What it is, however, is a struct (like the MinMaxGradient struct), that is used extensively in several different locations, and so I shall provide details on both of them up front.

The exact details I will provide on the next page. A common usage pattern of MinMaxCurves is:

```
namespace UnityEngine;
public struct Module {
    public ParticleSystem.MinMaxCurve curveField;
    public float curveFieldMultiplier;
}
```

While curveFieldMultiplier should not have any bearing on what's in curveField, and it appears that curveFieldMultiplier is applied afterwards, this is not always the case.
Recall that the ParticleSystem Modules are mere interfaces. curveFieldMultiplier serves as a way to quickly change aspects of the backend behind the interface. While you may set the actual curve to be constant, with a value of 0.0, if you set the Multiplier field to 1.0, you may override that constant value to 1.0.
The above is a true story, that happened in C# code, and all indications point to it happening in JSON too.

Moral of the story: Only set one - curve or multiplier. Never both.

```
namespace UnityEngine;
public struct ParticleSystem.MinMaxCurve {
    public ParticleSystemCurveMode mode;
    public float constant;
    public float constantMax;
    public float constantMin;
    public AnimationCurve curve;
    public AnimationCurve curveMax;
    public AnimationCurve curveMin;
    public float curveMultiplier;
}
public class AnimationCurve {
    public KeyFrame[] keys;
    public WrapMode postWrapMode;   // Normally see it set to 8
    public WrapMode preWrapMode;    // Normally see it set to 8
}
public struct KeyFrame {
    public float inTangent;     // Generally ignorable - not even sure if it works
    public float outTangent;    // ditto
    public float inWeight;      // Advanced usage only
    public float outWeight;     // ditto
    public float time;
    public float value;         // Normally see it set to 0

    public WeightedMode weightedMode;
}
public enum ParticleSystemCurveMode {
    Constant,       // Use the specified constant for the entire curve
    Curve,          // Use the specified curve for the curve
    TwoCurves,      // Use a random value between 2 curves for the curve
    TwoConstants    // Use a random value between 2 constants for the curve
}
public enum WrapMode {
    Once,           // the clip will stop when ended, time will be reset
    Loop,           // the clip will loop when ended
    PingPong,       // time will ping pong back and forth from beginning to end
    Default,        // read the default repeat mode set higher up
    ClampForever    // Plays back animation, keeps playing last frame at end forever
}
public enum WeightedMode {
    None,           // Exclude both weights when calculating curve
    In,             // Use inWeight when calculating prior curve segment
    Out,            // Use outWeight when calculating next curve segment
    Both            // Use both weights when calculating curve segments
}
```

You can find a set of useful prefabs here.
Something to note: The manual doesn't place any limits on time, but MniMaxGradient limits to 0.0 to 1.0
Hypothesis: 1.0 represents end of system duration. (Unconfirmed)
Important Note: if you wish to set a constant value, you can either directly set the MinMaxCurve to a float value, or do the same to the multiplier var.

While it's not technically correct, Unity will take care of it and set stuff appropriately.

5.2 The MinMaxGradient Struct

ParticleSystem.MinMaxGradient follows a similar construction to MinMaxCurve, as you can see below.

```
namespace UnityEngine;
public struct ParticleSystem.MinMaxGradient {
    public ParticleSystemGradientMode mode;

    public Color color;
    public Color colorMax;
    public Color colorMin;

    public Gradient gradient;
    public Gradient gradientMax;
    public Gradient gradientMin;
}
public Class Gradient {
    public GradientAlphaKey[] alphaKeys;
    public GradientColorKey[] colorKeys;
    public GradientMode mode;
}
public struct GradientAlphaKey {
    public float alpha;     // Alpha channel 0 - 1
    public float time;      // Confirmed 0 - 1
}
public struct ColorAlphaKey {
    public Color color;     // color at time
    public float time;      // Confirmed 0 - 1
}
public struct Color {
    public float r;
    public float g;
    public float b;
    public float a;
}
public enum ParticleSystemGradientMode {
    Color,          // Use a single color
    Gradient,       // Use a single gradient
    TwoColors,      // Use a random value between two colors
    TwoGradients,   // Use a random value between 2 gradients
    RandomColor     // Samples gradient (I think) at random times to select colors
}
public enum GradientMode {
    Blend,          // Find 2 adjacent keys, blend using lerp
    Fixed           // Return first key with time value > requested evaluation time
}
```

Notable differences include swapping out curves for gradients, but otherwise the same rules apply. A major addition is the introduction of the RandomColor mode, which theoretically samples random points in a provided gradient for the color. Also of note is that while Color holds an alpha key, these will be ignored in GradientColorKeys, as that role is provided by GradientAlphaKeys.
You can find a set of useful prefabs here.

5.3 The MainModule Struct

First a link to the Scripting API and the Manual.
The MainModule contains the bulk of the ParticleSystem configuration settings. Many of these settings, you will see directly in the root ParticleSystem component in Misc Mods JSON dumps. You are also able to modify many of these settings from the root ParticleSystem component. This is an artifact of migration to a new version, where both the old method (directly), and the new version (MainModule) are both supported.

I will use both formulations, depending on what prefabs/existing effects I copy, but make sure you only define any one parameter once. In addition to location changing, the interfaces have also changed. If you want to change a parameter as referenced in this manual, use the main module. If you just want to tweak something Misc Mods exported, do it in the root component.

That being said, I have placed the MainModule interface on the next page (I have ordered the more useful parameters at the top). For space, I shall impart knowledge in this list below on a parameter basis.

The most useful/important parameters are as below:

- loop/prewarm: For continuous effects, you may want to have the system loop its playing. Depending on how the particles appear, you may want to enable prewarm so it appears fully-formed.
- maxParticles: Fairly self-explanatory, max number of particles to track. 200-1000 is practical
- playOnAwake: Will you auto-play this system? Generally set to true
- duration: How long will this particle system last before it stops emitting?
- startColor: What's the initial particle color? Note that MinMaxGradient allows random colors
- startDelay: How long till the system starts emitting particles, once it starts playing
- startLifetime: How long each particle has to live, before it's erased
- startSpeed: Velocity of each particle on creation.
- startRotation: What orientation does the particle face on creation? Can be 3D
- startSize: What's the size of the particle when initialized? Can be 3D

Of the most advanced parameters, the ones most likely to be used are:

- simulationSpeed: Occasionally, you will want to change the simulation speed, as it is a faster way of tuning the system to fit into a given timeframe than modifying start speed/duration/lifetimes
- gravityModifier: Gravity does have some niche applications, i.e. weapon sparks, etc
- simulationSpace: Occasionally, you may want particles to move in world-space, so the effect doesn't always snap to the block, but instead leaves "echoes" in the world.

Some sample prefabs of several of these parameters are provided further down, here.

```
namespace UnityEngine;
public struct ParticleSystem.MainModule {
    public bool loop;           // Specifies whether the Particle System is looping.
    public int maxParticles;    // The maximum number of particles to emit.
    public bool playOnAwake;    // Automatically play this system?
    public bool prewarm;        // If loop, simulate a loop before it plays
                                // DOES NOT PLAY WELL WITH NON-0 START DELAY
    public float duration;      // The duration of the Particle System in seconds.

    public ParticleSystem.MinMaxGradient startColor;    // Initial particle color

    public ParticleSystem.MinMaxCurve startDelay;       //Start delay in seconds.
    public float startDelayMultiplier;

    public ParticleSystem.MinMaxCurve startLifetime;    //Particle lifetime in seconds.
    public float startLifetimeMultiplier;

    // Start Rotation parameters
    public bool startRotation3D;                        // A flag to specify individual axes calculation
    public ParticleSystem.MinMaxCurve startRotation;
    public float startRotationMultiplier;
    public ParticleSystem.MinMaxCurve startRotationX;
    public float startRotationXMultiplier;
    public ParticleSystem.MinMaxCurve startRotationY;
    public float startRotationYMultiplier;
    public ParticleSystem.MinMaxCurve startRotationZ;
    public float startRotationZMultiplier;

    // Start Size parameters
    public bool startSize3D;                            // A flag to specify individual axes calculation
    public ParticleSystem.MinMaxCurve startSize;
    public float startSizeMultiplier;
    public ParticleSystem.MinMaxCurve startSizeX;
    public float startSizeXMultiplier;
    public ParticleSystem.MinMaxCurve startSizeY;
    public float startSizeYMultiplier;
    public ParticleSystem.MinMaxCurve startSizeZ;
    public float startSizeZMultiplier;

    public ParticleSystem.MinMaxCurve startSpeed;
    public float startSpeedMultiplier;

    public ParticleSystemStopAction stopAction;         // ALWAYS SET TO NONE

    public float simulationSpeed;                       // Modifier for playback speed

    // Where are particles simulated?
    public ParticleSystemSimulationSpace simulationSpace;   // Only Local/World work
    public Transform customSimulationSpace;                 // Haven't gotten it to work

    // [ADVANCED] When true, use the unscaled delta time to simulate the Particle System.
    //            Otherwise, use the scaled delta time.
    public bool useUnscaledTime;

    // [ADVANCED] Handles how gravity affects particles
    public ParticleSystem.MinMaxCurve gravityModifier;
    public float gravityModifierMultiplier;

    // [ADVANCED] Control how the Particle System applies its Transform component to the particles it emits.
    public ParticleSystemScalingMode scalingMode;

    // [ADVANCED] Configure the Particle System to not kill its particles when their lifetimes are exceeded.
    public ParticleSystemRingBufferMode ringBufferMode;
    // [ADVANCED] When ParticleSystem.MainModule.ringBufferMode is set to loop, this value defines the
    //            proportion of the particle life that loops.
    public Vector2 ringBufferLoopRange;
    // [ADVANCED] Control how the Particle System calculates its velocity, when moving in the world.
    public ParticleSystemEmitterVelocityMode emitterVelocityMode;
    public float flipRotation;      // [ADVANCED] Makes some particles spin in the opposite direction. [0-1]
    // [ADVANCED] Configure whether the Particle System will still be simulated each frame, when offscreen.
    public ParticleSystemCullingMode cullingMode;
}
```

5.4 The EmissionModule Struct

Once more, I shall provide a link to the Scripting API and the Manual. For all that it is arguably the most essential module, it has a rather minimal accessible interface.

```
namespace UnityEngine;
public struct ParticleSystem.EmissionModule {
    public bool enabled;

    public ParticleSystem.MinMaxCurve rateOverDistance;
    public float rateOverDistanceMultiplier;

    public ParticleSystem.MinMaxCurve rateOverTime;
    public float rateOverTimeMultiplier;
}
```

Much as we shall see with *literally every other module*, there is an enabled bool, that toggles if this module is enabled. Outside of that, the EmissionModule has two parameters:

- rateOverDistance
- rateOverTime

Both of these parameters may or may not be set at the same time, and have separate functionalities. RateOverTime emits particles at the specified rates, at the specified times. RateOverDistance emits particles at the specified rates as the parent object travels the specified distances (useful for tire dust etc).

Those of you who read the Unity docs (or go through Misc Mods exports) will be aware that there is a "burst" component to this module. Unfortunately, said component is only available via script, or the Unity Inspector. You can roughly approximate a burst by setting the rateOverTime to ridiculous levels at a certain point in time, and clamping it down back to normal levels right before and after that point in time, but you will not be able to have a burst at the exact same time, unless you are using a referenced ParticleSystem that does have bursts.

Important Corollary: If you are referencing a ParticleSystem whose EmissionModule has a few bursts set, *you will not be able to change or disable them.*

For most purposes, a constant emission rate should be just fine. I have yet to experiment with changing rates over time, but I can see it being useful in the case of charging system particles.

5.5 The ShapeModule Struct

You know the drill. Script API, Manual
The ShapeModule covers how to modify the shape of the ParticleSystem emitter. Do you want to have particles form in a ring, donut, cone, etc.? This is the module where you define it. It is not as large as the MainModule, but oh lord is it annoying to describe.

One of the major feature of the ShapeModule is that it shares variables. Specific variables may have different functions depending on what ShapeType you've specified. In addition, some variables only have effects when in used in conjunction with certain shapes

On the page after next, I've compiled a list of the ShapeType enums that TT uses. When you see the exported variable in MiscMods, I'm pretty sure the corresponding index in said list is the actual ShapeType being used. Note how this, in no way lines up with the official Unity enums espoused in their documentation.

```
namespace UnityEngine;
public struct ParticleSystem.ShapeModule {
    public bool enabled;
    public ParticleSystemShapeType shapeType;

    // Common to most shape types (with obvious exceptions)
    public float radius;        // also used in Edge, funnily enough
    public float radiusThickness;
    public float arc;
    public ParticleSystemShapeMultiModeValue arcMode;
    public float arcSpread;
    public ParticleSystem.MinMaxCurve arcSpeed;
    public float arcSpeedMultiplier;

    // Cone shapes only
    public float angle;
    public float length;

    public Vector3 boxThickness;    // Box shape only
    public float donutRadius;       // Donut shape only

    // Edge shape only
    public ParticleSystemShapeMultiModeValue radiusMode;
    public float radiusSpread;
    public ParticleSystem.MinMaxCurve radiusSpeed;
    public float radiusSpeedMultiplier;

    // Mesh shape only
    public float normalOffset;
    public Mesh mesh;
    public bool useMeshMaterial;
    public int meshMaterialIndex;
    public SkinnedMeshRenderer skinnedMeshRenderer;
    public MeshRenderer meshRenderer;
    public ParticleSystemMeshShapeType meshShapeType;
    public bool useMeshColors;

    // Common to All
    public Vector3 position;
    public Vector3 rotation;
    public Vector3 scale;
    public bool alignToDirection;
    public float randomDirectionAmount;
    public float sphericalDirectionAmount;
    public float randomPositionAmount;
}
public enum ParticleSystemShapeMultiModeValue {
    Random,
    Loop,
    PingPong,
    BurstSpread
}
```

5.5.1 ShapeTypes Overview

the ShapeType enum is displayed below. While all are probably supported, I highly caution against using the obsolete options.

```
namespace UnityEngine;
public enum ParticleSystemShapeType {
    Sphere,
    [Obsolete] SphereShell,
    Hemisphere,
    [Obsolete] HemisphereShell,
    Cone,
    Box,
    Mesh,
    [Obsolete] ConeShell,
    ConeVolume,
    [Obsolete] ConeVolumeShell,
    Circle,
    [Obsolete] CircleEdge,
    SingleSidedEdge,
    MeshRenderer,
    SkinnedMeshRenderer,
    BoxShell,
    BoxEdge,
    Donut
}
```

DO NOT USE OBSOLETE OPTIONS.
THEIR FORMER FUNCTIONALITY HAS BEEN ROLLED INTO THE ShapeModule.radiusThickness VARIABLE.
IF YOU USE THE OBSOLETE MODULES, YOU INVITE AN ANCIENT EVIL INTO YOUR HOME: UNDEFINED BEHAVIOR.

In newer versions of the Unity docs, you will see options to apply texture tinting to particles, and new ShapeTypes. These are not available in TT. I've gone through what the game uses, and the above is what I have confirmed exists. I will reiterate that I have linked to the Unity Version Docs that I find it most likely that TT is using, since it has many of the same interfaces.

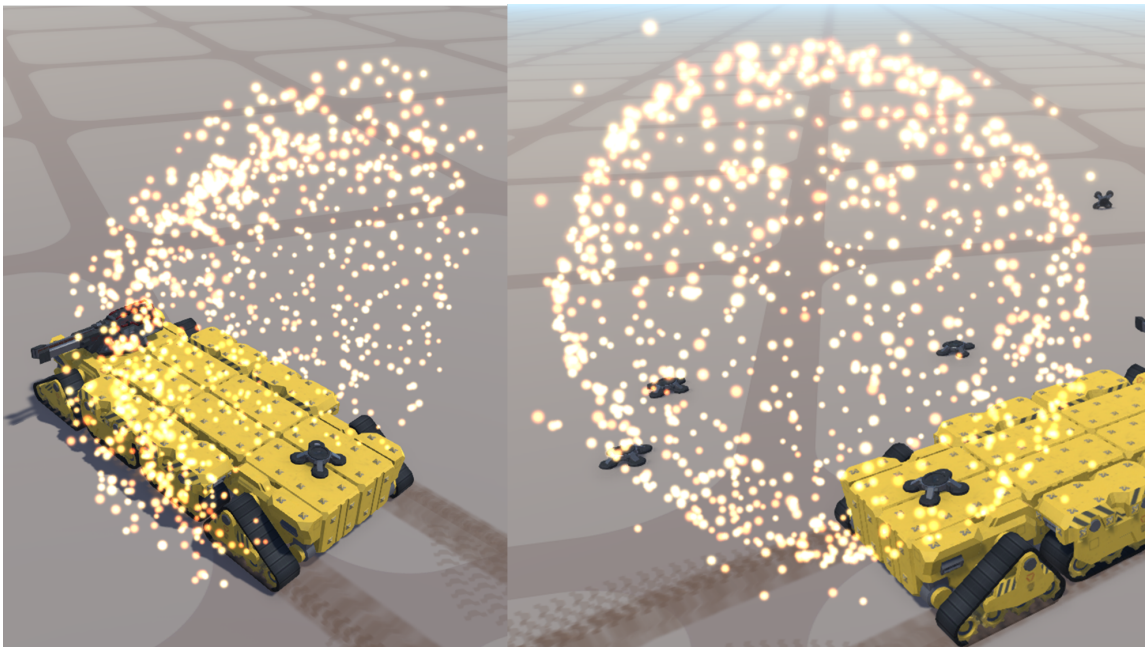
Even in the linked docs, you will find some of the newer arguments, e.g. textureBilinearFiltering. **However**, the vast majority of these cases are located within the Unity Manual pages. I believe that the existence of these fields in script form later is just the Unity developers taking extant features that relied on the Inspector, and making them available to scripts to use. Even though we are not writing C# scripts (well most of us), we are still impacted by this, since the JSON we write relies on the C# script interfaces, since the Block Injector is, after all, a C# tool.

In the following pages, I will elucidate exactly which parameters are used where. When necessary, I shall provide descriptions of what they do.

5.5.2 Sphere, Hemisphere

```
{
    "shape": {
        "shapeType": "Sphere",
        "radius": 1.0,
        "radiusThickness": 1.0,
        "scale": { "x": 1.0, "y": 1.0, "z": 1.0 },
        "rotation": { "x": 0.0, "y": 0.0, "z": 0.0 },
        "position": { "x": 0.0, "y": 0.0, "z": 0.0 },
        "alignDirection": false,
        "randomDirectionAmount": 0.0,
        "randomPositionAmount": 0.0
    }
}
```

- **radiusThickness** is a slider variable that goes from 0.0 to 1.0. When it is set to 0, particles will only begin on the surface of the shape. When it is set to 1, particles can start from anywhere within the shape volume. When set in between, a proportion of the volume can spawn particles
- **scale** is a quick and dirty way to change the size of the particle emitter
- **position** provides a way to offset the particle emitter
- **rotation** provides a way to modify the rotation (big surprise)
- **alignToDirection** tries to align particles to their initial direction of travel. It does not change *which* directions particles travel.
- **randomizeDirectionAmount** is another slider variable. When set to 0, it does nothing. When set to 1, particle direction is completely random
- **sphericalDirectionAmount** is useless on **Sphere** emitters
- **randomPositionAmount** is another slider variable. Once more, value of 1 is completely random. Spawning positions will remain bounded within the shape



5.5.3 Cone, ConeVolume

```
{
    "shape": {
        "shapeType": "Cone",
        "radius": 1.0,
        "radiusThickness": 1.0,
        "arc": 360.0,
        "arcMode": "Random",
        "arcSpread": 0.0,
        "arcSpeed": 1.0,
        "angle": 20.0,
        "length": 1.0,
        "scale": { "x": 1.0, "y": 1.0, "z": 1.0 },
        "rotation": { "x": 0.0, "y": 0.0, "z": 0.0 },
        "position": { "x": 0.0, "y": 0.0, "z": 0.0 },
        "alignDirection": false,
        "sphericalDirectionAmount": 0.0,
        "randomDirectionAmount": 0.0,
        "randomPositionAmount": 0.0
    }
}
```

- The particles diverge in proportion to their distance from the cone's center line.
- **sphericalDirectionAmount** will (since this is a non-sphere), blend directions towards the vector defined by the two points of the emitter center, and the particle spawn point
- **Cone** has all particles being emitted from the base of the cone described
- **ConeVolume** has particles being spawned from anywhere within the cone
- **angle** describes the angle of the cone. 0 describes a cylinder, while 90 gives a flat disc
- While it may seem **angle** only matters for **ConeVolume**, since **Cone** only emits from the cone base, **angle** also serves to determine the initial directions of particles, which affects both variants
- **length** describes the length of the cone (only matters for ConeVolume)

5.5.4 Box, BoxShell, BoxEdge

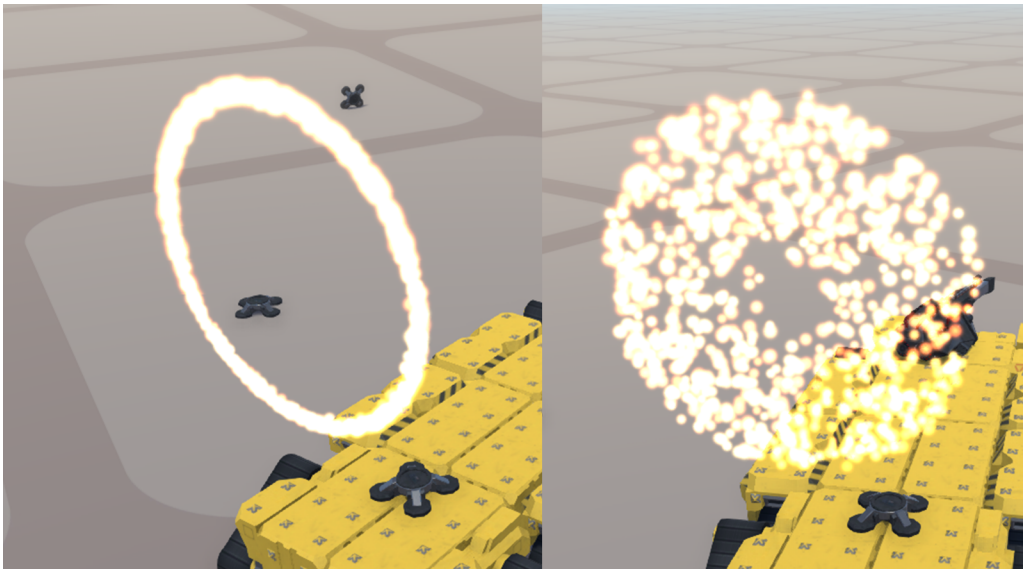
```
{
    "shape": {
        "shapeType": "Box",
        "scale": { "x": 1.0, "y": 1.0, "z": 1.0 },
        "rotation": { "x": 0.0, "y": 0.0, "z": 0.0 },
        "position": { "x": 0.0, "y": 0.0, "z": 0.0 },
        "alignDirection": false,
        "sphericalDirectionAmount": 0.0,
        "randomDirectionAmount": 0.0,
        "randomPositionAmount": 0.0
    }
}
```

- Box-type shapes emit particles from the edge, surface, or body of a box shape. The particles move in the emitter object's forward (Z) direction.
- In Misc Mods export, you will see a **box**field with Vector3 bounds. This is a deprecated field. You should be using **scale** to input box size parameters
- **shapeType** can be of variant **Box**, **BoxShell**, or **BoxEdge**.

5.5.5 Circle

```
{
    "shape": {
        "shapeType": "Circle",
        "radius": 1.0,
        "radiusThickness": 1.0,
        "arc": 360.0,
        "arcMode": "Random",
        "arcSpread": 0.0,
        "arcSpeed": 1.0,
        "scale": { "x": 1.0, "y": 1.0, "z": 1.0 },
        "rotation": { "x": 0.0, "y": 0.0, "z": 0.0 },
        "position": { "x": 0.0, "y": 0.0, "z": 0.0 },
        "alignDirection": false,
        "sphericalDirectionAmount": 0.0,
        "randomDirectionAmount": 0.0,
        "randomPositionAmount": 0.0
    }
}
```

- particles will, by default, move only in the plane of the circle
- (almost positive) that can be overriden by **sphericalDirectionAmount** and **randomDirectionAmount**
- **radiusThickness** controls whether particles emit from circle edge, or entire area



5.5.6 SingleSidedEdge

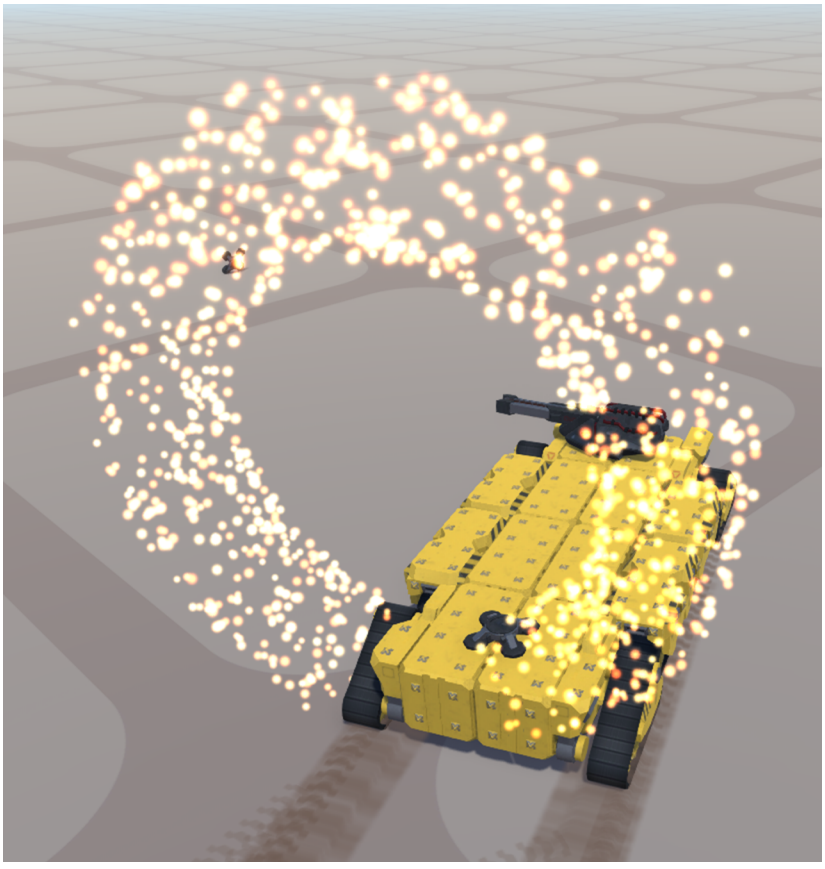
```
{
    "shape": {
        "shapeType": "SingleSidedEdge",
        "radius": 1.0,
        "radiusMode": "Random",
        "radiusSpread": 0.0,
        "radiusSpeed": 1.0,
        "scale": { "x": 1.0, "y": 1.0, "z": 1.0 },
        "rotation": { "x": 0.0, "y": 0.0, "z": 0.0 },
        "position": { "x": 0.0, "y": 0.0, "z": 0.0 },
        "alignDirection": false,
        "sphericalDirectionAmount": 0.0,
        "randomDirectionAmount": 0.0,
        "randomPositionAmount": 0.0
    }
}
```

- particles will move in the emitter's upward (Y) direction.
- **radius**, for the **SingleSidedEdge** shape type is repurposed, and describes the length of the edge instead
- **radiusMode**, **radiusSpread**, **radiusSpeed**, and **radiusSpeedMultiplier** perform identically to their arc-focused counterparts, just here they do the same thing along the line.
- Yes, **radiusSpeedMultiplier** and **arcSpeedMultiplier** exist, no, I haven't bothered to set them in these prefabs

5.5.7 Donut

```
{
    "shape": {
        "shapeType": "Donut",
        "donutRadius": 0.15,
        "radius": 1.0,
        "radiusThickness": 1.0,
        "arc": 360.0,
        "arcMode": "Random",
        "arcSpread": 0.0,
        "arcSpeed": 1.0,
        "scale": { "x": 1.0, "y": 1.0, "z": 1.0 },
        "rotation": { "x": 0.0, "y": 0.0, "z": 0.0 },
        "position": { "x": 0.0, "y": 0.0, "z": 0.0 },
        "alignDirection": false,
        "sphericalDirectionAmount": 0.0,
        "randomDirectionAmount": 0.0,
        "randomPositionAmount": 0.0
    }
}
```

- Particles will move outwards from the ring of the Torus (appear perpendicular to the surface, I think) by default
- **donutRadius** controls the thickness of the ring, whereas **radius** controls the radius of the ring



5.5.8 Mesh, MeshRenderer, SkinnedMeshRenderer

```
{
    "shape": {
        "shapeType": "Mesh",
        "normalOffset": 0.0,
        "mesh": null,
        "useMeshMaterial": false,
        "meshMaterialIndex": 0,
        "skinnedMeshRenderer": false,
        "meshRenderer": false,
        "meshShapeType": "Vertex",
        "useMeshColors": false,
        "scale": { "x": 1.0, "y": 1.0, "z": 1.0 },
        "rotation": { "x": 0.0, "y": 0.0, "z": 0.0 },
        "position": { "x": 0.0, "y": 0.0, "z": 0.0 },
        "alignDirection": false,
        "sphericalDirectionAmount": 0.0,
        "randomDirectionAmount": 0.0,
        "randomPositionAmount": 0.0
    }
}
```

- While you can technically use these shape types, I don't know how they work. so yeah - use at your own peril.
- Will note that **MeshRenderers** work similar to **ParticleSystemRenderers**. Need a **MeshFilter** or **TextMesh** on same **GameObject** to work

```
namespace UnityEngine;
public enum ParticleSystemMeshShapeType { Vertex, Edge, Triangle }
public class MeshRenderer {
    public Mesh additionalVertexStreams;
}
public class MeshFilter {
    public Mesh mesh;
    public Mesh sharedMesh;
}
public class TextMesh {
    public string text;
    public TextAlignment alignment; // Left, Right, Center
    public TextAnchor anchor;
    public float characterSize;
    public Color color;
    public Font font;
    public int fontSize;        // for use with dynamic fonts
    public FontStyle fontStyle; // [Normal, Bold, Italic, BoldAndItalic]
    public float lineSpacing;
    public float offsetZ;
    public bool richText;       // enable HTML-style tags for text
    public float tabSize;
}
```

5.6 The ColorOverLifetimeModule Struct

You know the drill. Script API, Manual

```
namespace UnityEngine;
public struct ParticleSystem.ColorOverLifetimeModule {
}
```

5.7 The ColorBySpeedModule Struct

You know the drill. Script API, Manual

```
namespace UnityEngine;
public struct ParticleSystem.ColorBySpeedModule {
}
```

5.8 The RotationOverLifetimeModule Struct

You know the drill. Script API, Manual

```
namespace UnityEngine;
public struct ParticleSystem.RotationOverLifetimeModule {
}
```

5.9 The RotationBySpeedModule Struct

You know the drill. Script API, Manual

```
namespace UnityEngine;
public struct ParticleSystem.RotationBySpeedModule {
}
```

5.10 The SizeOverLifetimeModule Struct

You know the drill. Script API, Manual

```
namespace UnityEngine;
public struct ParticleSystem.SizeOverLifetimeModule {
}
```

5.11 The SizeBySpeedModule Struct

You know the drill. Script API, Manual

```
namespace UnityEngine;
public struct ParticleSystem.SizeBySpeedModule {
}
```

5.12 The InheritVelocityModule Struct

You know the drill. Script API, Manual

```
namespace UnityEngine;
public struct ParticleSystem.InheritVelocityModule {
}
```

5.13 The VelocityOverLifetimeModule Struct

You know the drill. Script API, Manual

```
namespace UnityEngine;
public struct ParticleSystem.VelocityOverLifetimeModule {
}
```

5.14 The LimitVelocityOverLifetimeModule Struct

You know the drill. Script API, Manual

```
namespace UnityEngine;
public struct ParticleSystem.LimitVelocityOverLifetimeModule {
}
```

5.15 The LightsModule Struct

You know the drill. Script API, Manual

```
namespace UnityEngine;
public struct ParticleSystem.LightsModule {
    public bool enabled;                // is this module enabled?

    public Light light;                 // set a reference to the light prefab
    public int maxLights;               // set a max number of lights it can create
    public float ratio;                 // what ratio of particles will get a light

    // Define a curve to apply custom intensity scaling to particle lights
    public ParticleSystem.MinMaxCurve intensity;
    public float intensityMultiplier;

    // Define a curve to apply custom range scaling to particle lights
    public ParticleSystem.MinMaxCurve range;
    public float rangeMultiplier;

    public bool sizeAffectsRange;       // does particle size affect final intensity
    public bool alphaAffectsIntensity;  // does particle alpha affect final intensity

    public bool useParticleColor;       // Multiply light color by particle color
    public bool useRandomDistribution;  // Randomly decide which particles get lights

}
```

The LightsModule is relatively simple as modules go. It has the standard enabled boolean flag, a max size (number of created objects) and ratio of applied particles for performance reasons, and a set of multipliers to change its aspects.
Every single property in the LightsModule struct may be of use to modders at one point or another, so I won't pick and choose, only offer the old warning to be careful when using the MinMaxCurves and associated multipliers.
The essential core of this Module would be the UnityEngine.Light object - the light you want to have your particles emit. The components of the interface we are able to modify have been provided on the next page, but you may find the official manual a useful starter guide on what does what.

5.15.1 The UnityEngine.Light System

```
public class Light {
    public LightType type;          //The type of the light
    public Color color;             //The color of the light.
    public float colorTemperature;
    // The color temperature of the light. Correlated Color Temperature
    // (abbreviated as CCT) is multiplied with the color filter when calculating the final color
    // of a light source. The color temperature of the electromagnetic radiation emitted from
    // an ideal black body is defined as its surface temperature in Kelvin. White is 6500K
    // according to the D65 standard. Candle light is 1800K. If you want to use lightsUseCCT,
    // lightsUseLinearIntensity has to be enabled to ensure physically correct output.
    // See Also: GraphicsSettings.lightsUseLinearIntensity, GraphicsSettings.lightsUseCCT.
    public float spotAngle;         //The angle of the light's spotlight cone in degrees.

    public float range;             //The range of the light.
    public float intensity;         //The Intensity of a light is multiplied with the Light color.

    public LightRenderMode renderMode;  //How to render the light.
    public Flare flare;                 //The flare asset to use for this light.
                                        //Unfortunately, we have no way to create new flares,
                                        //since they need the Unity inspector to setup.
                                        //Thus, referencing any ones you find will have to suffice

    public Vector2 areaSize;        //The size of the area light.
    public float bounceIntensity;   //The multiplier that defines the strength of the bounce lighting.

    public LightBakingOutput bakingOutput;      // This property describes the output of the last
                                                // Global Illumination bake.
    public LightmapBakeType lightmapBakeType;   // This property describes what part of a light's
                                                // contribution can be baked.

    public Texture cookie;      //The cookie texture projected by the light.
    public float cookieSize;    //The size of a directional light's cookie.

    public int cullingMask; //This is used to light certain objects in the scene selectively.

    //Shadows
    public LightShadows shadows;                //How this light casts shadows
    public float shadowBias;                    //Shadow mapping constant bias.
    public int shadowCustomResolution;          //The custom resolution of the shadow map.
    public float shadowNearPlane;               //Near plane value to use for shadow frustums.
    public float shadowNormalBias;              //Shadow mapping normal-based bias.
    public Rendering.LightShadowResolution shadowResolution;  //The resolution of the shadow map.
    public float shadowStrength;                //Strength of light's shadows
}
```

The struct is, as usual, relatively self-explanatory (at least with my added comments). You will find a reference to the classes and enums mentioned above on the next page. The one exception: the Flare class. While I have left it in since it is technically possible to reference an existing Flare, Flares are another thing meant only to be set in the UnityEditor, and thus cannot be created ex nihilo.

```
namespace UnityEngine;
public struct LightBakingOutput {
    // is light contribution already stored in lightmaps/probes?
    public bool isBaked;
    // What part of a light's contribution was baked?
    public LightmapBakeType lightmapBakeType;

    // In case of LightmapBakeType.Mixed:
    // What Mixed mode was used to bake the light?
    public MixedLightingMode mixedLightingMode;
    // Contains index of occlusion mask channel if any, -1 otherwise
    public int occlusionMaskChannel;
    // contains index of light as seen from occlusion probes pov if any, -1 otherwise
    public int probeOcclusionLightIndex
}
public enum LightType {
    Spot,           // Spot light (cone shaped lights)
    Directional,    // Directional Light (infinitely-far away spotlights)
    Point,          // Point light (sends light in all directions equally)
    Area            // Area light - lightmap/lightprobe only
}
public enum LightmapBakeType {
    Realtime,       // No baking. Dynamic all the way (inadvisable if baking is possible)
    Baked,          // Yes baking. Everything baked
    Mixed           // Mixed, depends on the exact setting in MixedLightingMode
}
public enum MixedLightingMode {
    IndirectOnly,   // Realtime direct lighting, while indirect light is baked
    Shadowmask,     // same as above, Shadowmasks and probe occlusion are made
                    // for baked shadows.
    Subtractive     // baked direct and indirect for static. Dynamic are not baked
}
public enum LightRenderMode {
    Auto,           // Automatically choose
    ForcePixel,     // Force it into a pixel light
    ForceVertex     // Force it into a vertex light
}
public enum LightShadows {
    None,           // No shadows (default)
    Hard,           // cast "hard" shadows (no filtering)
    Soft            // cast soft shadows (4x PCF filtering)
}
public enum Rendering.LightShadowResolution {
    FromQualitySettings,    // Use QualitySettings default
    Low,
    Medium,
    High,
    VeryHigh
}
```

5.16 The NoiseModule Struct

You know the drill. Script API, Manual

```
namespace UnityEngine;
public struct ParticleSystem.NoiseModule {
}
```

5.17 The TrailModule Struct

You know the drill. Script API, Manual

```
namespace UnityEngine;
public struct ParticleSystem.TrailModule {
}
```

5.18 The CollisionModule Struct

You know the drill. Script API, Manual

```
namespace UnityEngine;
public struct ParticleSystem.CollisionModule {
}
```

A Useful Class Prefabs

A.1 MinMaxCurve

Below, are a series of useful prefabs for MinMaxCurves. You will note that I have provided no prefab for TwoCurves. That is because it would not fit on a single page

```
{
    "mode": "Constant",
    "constant": 1.0,
    "curve": null,
    "curveMin": null,
    "curveMax": null
}
```

```
{
    "mode": "TwoConstants",
    "constantMin": 1.0,
    "constantMax": 1.0,
    "curve": null,
    "curveMin": null,
    "curveMax": null
}
```

```
{
    "mode": "Curve",
    "curve": {
        "keys": [
            {
                "time": 0.0,
                "value": 1.0,
                "weightedMode": "None"
            },
            {
                "time": 0.5,
                "value": 2.0,
                "weightedMode": "None"
            },
            {
                "time": 1.0,
                "value": 1.0,
                "weightedMode": "None"
            },
        ],
        "postWrapMode": "Loop",
        "preWrapMode": "Default"
    },
    "curveMin": null,
    "curveMax": null
}
```

A.2 Transform

Below, I provide a sample Transform module that has everything you would need. I stuck it here for spacing reasons

```
{
    "UnityEngine.Transform": {
        "localScale": { "x": 1.0, "y": 1.0, "z": 1.0 },
        "localEulerAngles": { "x": 0.0, "y": 0.0, "z": 0.0 },
        "localPosition": { "x": 0.0, "y": 0.0, "z": 0.0 }
    }
}
```

A.3 MinMaxGradient

Below, are a series of useful prefabs for MinMaxGradients. You will note I have not provided a prefab for TwoGradients, for the same reason I have not provided a prefab for TwoCurves: it's just too big to fit on a single page.

```
{
    "mode": "Color",
    "color": {
        "r": 1.0,
        "g": 0.2,
        "b": 0.3,
        "a": 1.0
    },
    "gradient": null,
    "gradientMin": null,
    "gradientMax": null
}
```

```
{
    "mode": "TwoColors",
    "colorMin": {
        "r": 1.0,
        "g": 0.2,
        "b": 0.1,
        "a": 1.0
    },
    "colorMax": {
        "r": 1.0,
        "g": 0.6,
        "b": 0.4,
        "a": 1.0
    },
    "gradient": null,
    "gradientMin": null,
    "gradientMax": null
}
```

A prefab for "Gradient" mode has been provided on the next page. The "RandomColor" mode should use a similar construction, just swap out "mode": "Gradient" to "mode": "RandomColor".

```
{
    "mode": "Gradient",
    "gradient": {
        "alphaKeys": [
            {
                "time": 0.0,
                "alpha": 1.0,
            },
            {
                "time": 0.5,
                "alpha": 2.0,
            },
            {
                "time": 1.0,
                "alpha": 1.0,
            },
        ],
        "colorKeys": [
            {
                "color": {
                    "r": 1.0,
                    "g": 0.2,
                    "b": 0.3,
                },
                "time": 0.0
            },
            {
                "color": {
                    "r": 1.0,
                    "g": 0.6,
                    "b": 0.3
                },
                "time": 0.5
            },
            {
                "color": {
                    "r": 1.0,
                    "g": 0.2,
                    "b": 0.3,
                },
                "time": 1.0
            }
        ],
        "mode": "Blend",
    },
    "gradientMin": null,
    "gradientMax": null
}
```

A.4 MainModule Parameters

Below, you will find useful parameters for the MainModule struct, which is described above.

B Sample Tricks

B.1 Lightning

B.2 Scifi Symbols

C Sample References