OULUN AMMATTIKORKEAKOULU

Joona Raappana

Great Particles and How to Make Them

Development of guidelines for the Unity Particle System

Great Particles and how to make them

Development of guidelines for the Unity Particle System

Joona Raappana
Great Particles and how to make them
Syksy 2018
Tietojenkäsittely
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietojenkäsittelyn koulutusohjelma, Web-sovelluskehitys

Tekijä(t): Joona Raappana
Opinnäytetyön nimi: Great Particles and how to make them
Työn ohjaaja: Matti Viitala
Työn valmistumislukukausi ja -vuosi: Syksy 2018 Sivumäärä: 60 + 3

Tämän opinnäytetyön tarkoituksena oli käydä läpi partikkeleiden valmistusta Unity pelimoottorissa. Tavoitteena oli luoda kattava katsaus partikkelisysteemin osiin ja sen käyttöön visuaalisten efektien luomisessa ja samalla oppia itse enemmän kyseisestä aiheesta. Työ tehtiin tilaustyönä Oulu Game LAB:lle, koska kyseisestä aiheesta ei ole tarpeeksi ohjausta.

Haasteen työssä oli avata partikkelisysteemin ovia tarpeeksi uusille ihmisille, ilman että mentäisiin liian syvälle, ja samalla oppia itse tarpeeksi kertomaan aiheesta.

Työssä käytettiin apuna Unityn omaa dokumentaatiota sekä internetistä löytyviä materiaaleja täydentämään aikaisempaa tietoa.

Aluksi työssä käydään läpi erilaisia visuaalisia efektejä ja sen jälkeen kerrotaan partikkeleiden rakenteesta ja visuaalisista efekteistä Unityssä. Eniten työssä keskitytään Unityn partikkelisysteemin eri moduuleihin ja niiden käyttötarkoituksiin tehdessä efektejä. Työssä kuvataan myös, mitä partikkelisysteemin ohessa voi käyttää ja painotetaan optimoinnin tärkeyttä.

Lopuksi näytetään esimerkkien avulla tilanteita, joissa partikkelit toimivat hyvin ja huonosti. Esimerkeillä myös demonstroidaan, kuinka eri pelien efektejä voi hajottaa osiin ja rakentaa itse.

Työ edisti osaamistani valtavasti, ja vaikka paljon jouduttiin leikkaamaan pois aiheen laajuuden vuoksi, niin olen lopputulokseen tyytyväinen ja toivon että tulevat Game LAB opiskelijat saavat tästä jotain irti.

Asiasanat: peligrafiikka, peliala, pelisuunnittelu, tehosteet

ABSTRACT

Oulu University of Applied Sciences
Degree programme in Business Information Systems, Web software development

---

Author(s): Joona Raappana
Title of thesis: Great Particles and How to Make Them
Supervisor(s): Matti Viitala
Term and year when the thesis was submitted: Fall 2018 Number of pages: 60 + 3

The purpose of this thesis was to go through the manufacturing process of particles in the Unity game-engine. The goal was to create a comprehensive overview to the parts of the Particle System and its uses in creating visual effects, while learning more about the subject myself. The thesis was made as an order to Oulu Game LAB, because their guidance on the subject was lacking.

The challenge of this work was to open the doors to the particle system to new users, while not diving too deep, and at the same time learn enough to tell about the subject.

During its making, Unity's own documentation and materials found on the internet were used to bolster previous knowledge of the system.

At first, the thesis covers different kinds of visual effects, after which the reader is told about the structure of the particles and visual effects in Unity. Most of all, the thesis focuses on Unity's particle system and its modules and their uses in the making of effects. The work also describes, what you can use in conjunction with the system and emphasizes the importance of optimization.

Finally, we display cases where particles work well or bad with the use of examples. The examples are also used to demonstrate, how different effects in games can be broken down and replicated.

The thesis advanced my skills greatly and even though a lot of content had to be cut due to the sheer scope of the subject, I am overall happy with the end result and hope that future Game LAB students can get something out of this.

Keywords: Effects, Game graphics, Game industry, Game design

TABLE OF CONTENTS

1 PREFACE

While the Particle System isn't something that is absolutely necessary to Game Development, it is used to enhance the feel of the game, to make it feel more immersive, which is a big point during the later stages of development. As the System is intimidating to approach to say the least, I decided to combine my want to learn the System with my thesis.

Oulu Game LAB is an institution which simulates working in the game industry and does it well with its coaching of various subjects and lectures, but there is no "guide" to effects or particles. While there is a Unity coach that can be called to help with Unity related problems, he cannot be there at every moment hence the creation of this thesis.

The version that was used in the creation of the thesis is Unity 2018.1.2f1, this is because they added the Orbital velocity in that version. Also, they made numerous smaller improvements that I wanted to use in the making of the thesis. I also had this specific version installed and had a few particles made beforehand that I wanted to use in the Scene that I will give to Oulu Game LAB to use and changing versions is known to cause problems both unexpected and otherwise, so I refrained from doing so.

The aim of the thesis is to work as a guide to the Particle System in Unity 2018.1.2f1 and help ease the making of particles for the games made in Oulu Game LAB. And while there are plenty of guides in YouTube and other platforms, the thesis is meant to compile the necessary information in a single place with examples of how they work.

During the making of the thesis, I also held a lecture about particles in Game LAB. Based on the feedback of the lecture, the structure of this thesis was changed drastically. Most parts were reduced to cover less and instead more emphasis was put on the basics.

A big thank you to Oulu Game LAB for letting me use the premise when I work on this thesis. Also, a huge thank you to the following YouTubers, Mirza, Gabriel Aguiar Prod and DucVu Fx for making publicly available guides to beautiful effects, for free no less.

2 VISUAL EFFECTS

2.1 Types of effects

Visual Effects in games consist mainly of gameplay effects (power ups, magic, sword trails) and environmental effects (rain, smoke). Different games have different amount of each, for example a game like Tekken will have more emphasis on gameplay effects, while a story-driven game may emphasize environmental effects more.

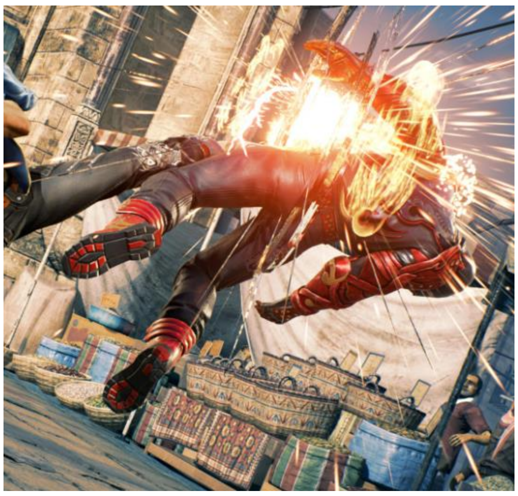


FIGURE 1 Tekken 7 hit effect (Namco Bandai Games, 2017)

Gameplay effects center around mechanics and as such require understanding of said mechanics and by extension, cooperation with designers. Gameplay effects consist of hit effects and spells to name a few and they are usually short-lived but flashy effects as seen in figure 1.

Environmental effects on the other hand requires the cooperation of other artists in the team, so as to fit the effects properly to the game's art style. Environmental effects tend to be subtler than gameplay effects and focus on creating the mood or feel for the game. These effects consist of smoke, rain, cloud for example. As seen in in figure 2, they don't necessarily have to be super realistic, depending on the game of course.

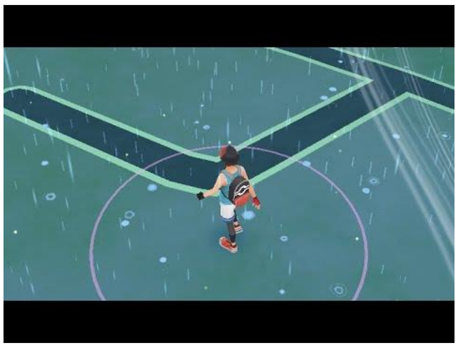


FIGURE 2 Pokémon GO environmental effect (Niantic, 2018)

Another way to compartmentalize Visual Effects is as described in the quote from Matt Schwartz below. Realistic effects comprise of highly detailed effects found in the real world, smoke and water are prime examples of such effects. Also, they often require more work, since they are grounded in reality and mimic an event from the real world.

> *"Realistic effects aim to reproduce detail found in real-world scenarios while stylized effects use fewer small details and focus more on shape color and form." (Matt Schwartz 2017 cited 24.11.2018).*

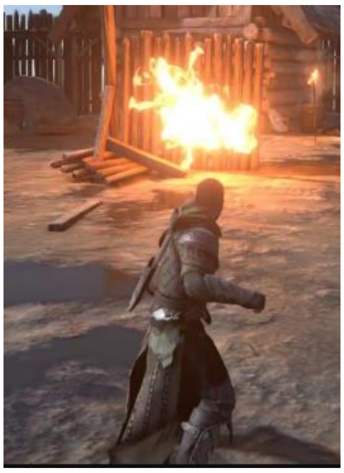


FIGURE 3 Realistic fire from Realistic Effects Pack 4 (Kripto289, 2017)

Stylized effects on the other hand are more over-the-top and less detailed, used to represent things that are not bound by reality like magic and hit effects. In figure 4, one can see an example of a stylized effect from Kingdom Hearts. Stylized effects are also often hand drawn to exaggerate the characteristics of the texture.

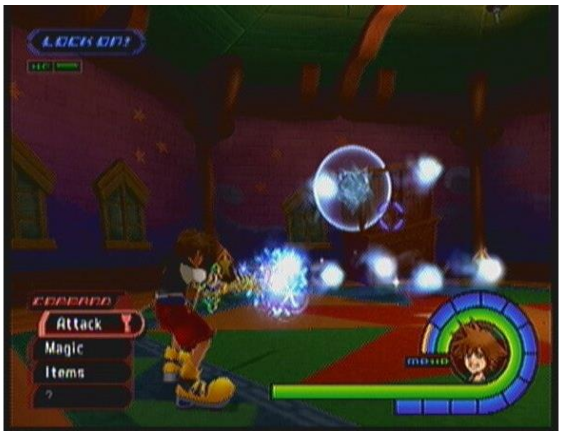


FIGURE 4 Stylized blizzard effect (Square Enix, 2002)

When making Visual Effects, regardless of type, the developer must make sure that the effect feels natural in the game. Nothing breaks a player's immersion quite as well, as an effect that covers most of their field of view and feels completely out of place. The best effects are the ones, that the player feels should be there.

2.2 Visual Effects in Unity

"Particle systems are like salt; just a small amount can add extra "pizzazz" to whatever you're cooking up. Modern games that don't use particle systems in some manner can feel quite bland." (Anthony Uccello 2018, cited 26.11.2018)

One way of creating VFX (Visual Effects) in Unity is by using the Particle System. The Particle System emits particles which are small representations of simple images or meshes, that make up a larger whole. Particles are used when the desired effect or event is too difficult to do by using meshes or sprites, the desired effect usually being something that has no defined shape, or a fluid of some kind as shown in figure 5.

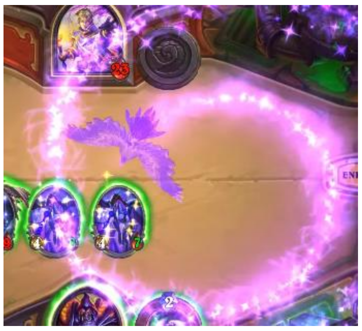


FIGURE 5 Medivh effect from Hearthstone (Blizzard Entertainment, 2017)

With these particles one can believably simulate effects that give life to their game and increases immersion for the player. To create believable effects, usually more than one particle system is layered in the effect, as the things one can do with just one are quite limiting.

In the Particle System window, one sets the modules of the particles that affect their behavior, these can range from collisions to speed to the color changes over their lifetime. The Particle System in Unity is component, meaning it can be added to any gameobject in the scene, this makes using the system flexible and easy.

While working on this, Unity announced the new **Visual Effect Graph** coming in Unity 2018.3. The new tool seems to combine particles with other visual effects into a node-based system. It is not the focus of this thesis, but it does look amazing.

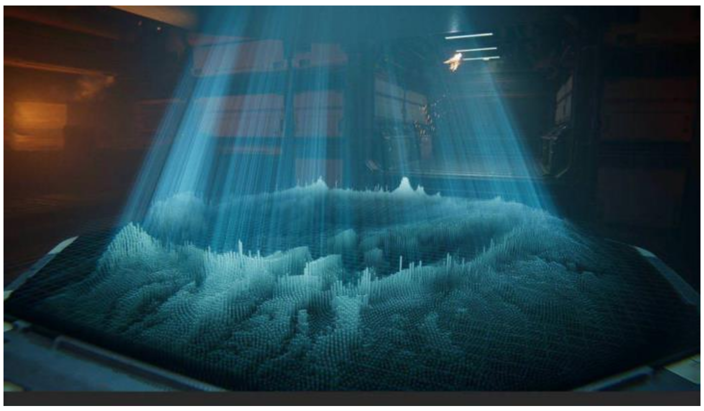


FIGURE 6 Made with Visual Effect Graph (Unity Technologies, 2018)

2.2.1 Textures

Textures make up the shape of the particle and in some cases the color of it as well. Textures are made in an image editing software, usually Photoshop. In the texture one wants to make the general shape of their particle, whether they decide to emit multiples or just one. Usually one leaves the texture white for recoloring in Unity itself, but in specific cases they can color it beforehand.

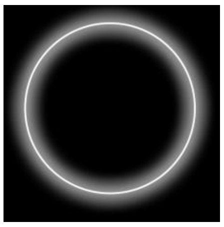


FIGURE 7 Glowing ring texture

Another way to make textures is to make them in a symmetrical grid known as a **Texture Sheet**. This can either mean a smaller grid (2x2) that holds four different shapes for a more specific use explained later in the optimization chapter of the thesis. One can also make a bigger grid (8x8) that holds all the frames of an animation that can be then played in each particle.

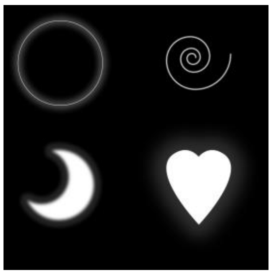


FIGURE 8 Texture with 4 different shapes

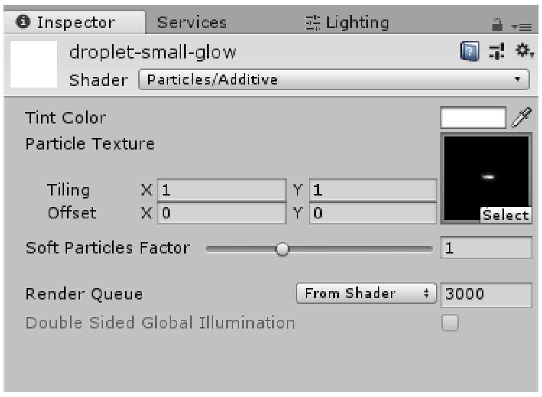
One should save the textures as a **JPEG** file, if they wish to conserve the file size and if the particle requires no transparency. **PNG** on the other hand should be used on all textures that will use the **Alpha** channel to manipulate transparency.

For sizes, one should work at a higher resolution like 2048x2048 as one can always downscale in Unity, but not scale the textures up. Textures should also be kept at a size in the power of two, to not require rescaling when the game is built. The highest size for PC games is 4096x4096 and for mobile 1024x1024, but 2048x2048 is recommended for PC, since many cards do not support higher sizes.

2.2.2 Materials

Materials control how the surface of an object should be rendered, whether that be a flat 2D particle or a complex mesh. Materials can have textures attached to them or just be a plain color, the options present for the material depend on the shader used on it. For this thesis, we've narrowed it down to only the shaders under the "Particles" sub-category.

The **Additive** shader, this one is used often, since it creates a glow on the particles and a lot of particles want that functionality. How the Additive shader works is it adds the color values of the particle to the background, so dark areas of the texture disappear leaving only the area that is white in the texture. Do keep in mind that this means if the particle is against a bright background, it will start to disappear.



droplet-small-glow
Shader Particles/Additive

Tint Color
Particle Texture

Tiling X 1 Y 1
Offset X 0 Y 0

Soft Particles Factor 1

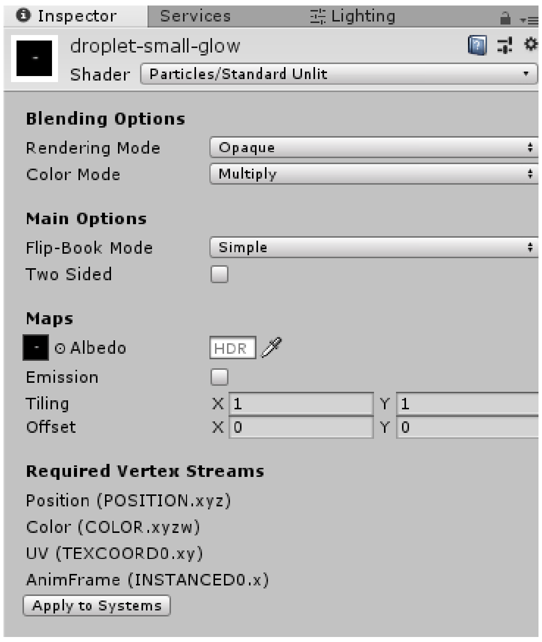
Render Queue From Shader 3000
Double Sided Global Illumination

FIGURE 9 Material options

For options the ones to keep an eye on is the **Soft Particles** factor and **Tint Color**. Soft Particles factor controls whether the particles blend with intersecting objects in the scene, the higher the value the better they blend with other objects. One should save their materials with 1, 2 and 3 soft particles factor to see which works best for them. Tint color adds a color on top of the texture, by default this is set to grey which is completely fine and if one decides to change it, they need only keep in mind the fact that it affects the color one sets in the Particle System window.

Alpha Blended takes the color of the background and the color of the particle and takes the aver age of the two using the Alpha value of the particle. Useful for darker effects such as smoke. Ordering must be used for consistent appearance though, otherwise one ends up with a really choppy and jarring effect. It also shares the options with the Additive shader.

Unlit shaders have no lighting calculations, so they are the best for particles that aren't supposed to be affected by light or shadows. **Lit** shaders on the other hand take all of this into account.



droplet-small-glow

Shader: Particles/Standard Unlit

Blending Options

Rendering Mode: Opaque

Color Mode: Multiply

Main Options

Flip-Book Mode: Simple

Two Sided: ☐

Maps

Albedo: HDR

Emission: ☐

Tiling: X 1 Y 1

Offset: X 0 Y 0

Required Vertex Streams

Position (POSITION.xyz)

Color (COLOR.xyzw)

UV (TEXCOORD0.xy)

AnimFrame (INSTANCED0.x)

FIGURE 10 Unlit and Lit options

3 THE PARTICLE SYSTEM

Now we arrive at the main dish of the thesis, the Particle System itself. This part will consist of explanations of every single module in the system and where to use them and how they affect each other. Outlining each one of the interactions between modules is way out of scope but I will outline the module in an overview and list some of the notable fields that one should be aware of when using them.

Most particles look just fine with the 4 “default” modules alone, yet it is with the Over Lifetime modules and the other modules that one really brings the particle to life. It is no longer a jarring block or spray of color, but rather something that blends in to the environment and doesn’t catch the players eye as something that doesn’t belong there in the first place.

Also, before one begins making any systems, one should make an empty particle system as a parent object, so they can play all effects under the parent system when one selects it as shown in figure 11. Empty means that one has turned off all modules, rendering the system incapable of emitting anything.

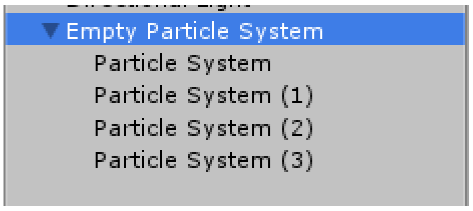


FIGURE 11 Parent system setup

3.1 Constants and Curves

In the Particle System, a lot of the fields in modules have a numerical value that one can set to control the designated effect. These values do not have to be **Constants**, by clicking the small triangle as shown in figure 12, one can set certain fields to either be **Curve**, **Random between two Constants** or **Random between two Curves** instead. This will help in getting a variety of behaviors from a single field.

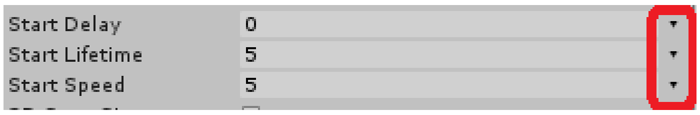

Start Delay 0
Start Lifetime 5
Start Speed 5

FIGURE 12 Settings change button

It is up to the user which one to apply to their fields and often experimenting with different settings leads to unexpected, but interesting results. As such, one shouldn't take the words below as gospel, but rather find what works for them and their desired effect.

Constant means that the value of that certain property will not change in magnitude during the lifetime of the particle. As such, this setting is best used with values that one does not desire to change at any point.

Curve allows one to implement a behavior that follows the specified curve in the field. For example, one might make a particle grow, then reduce the size and finish off by making it twice the size. Curves are therefore handy in situations, where one wants multiple modifications to the value over the specified parameters.

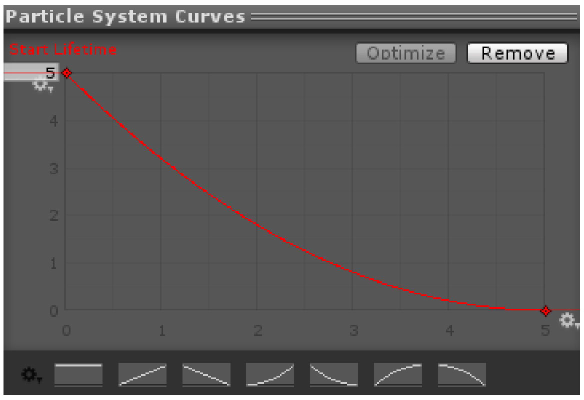


FIGURE 13 Example curve

Random between two Constants allows the developer to add randomness to their values, by having the sizes of the particles range from five to ten on emission for example. This is an often-used setting, due to the nature of particles being chaotic and usually wanting to look more lifelike.

Random between two Curves gives one the opportunity to control two curves in the editor that determine the range of values the particle may have. Unity chooses a curve that fits between the range specified, so one needs to think of this setting as controlling the ranges of values that they want the particle to adhere to. This setting is used for the reason as **Random between two Constants**, to add more liveliness and chaos to the effect.

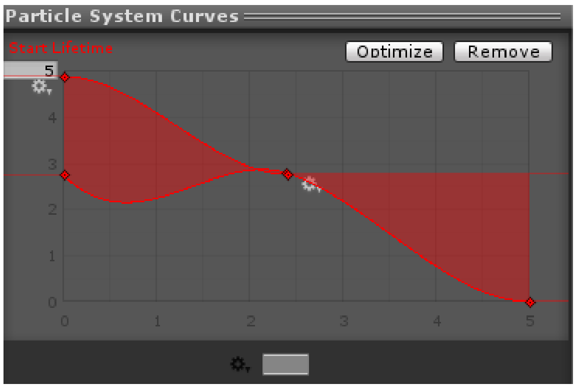


FIGURE 14 Example of Random between two Curves setup

As a final note, saving curves in the curve editor list saves a ton of time in future endeavors. If one plans on making a lot of particles, they should save the curves for future use. Each field has their unique curve lists, so saving in one doesn't make it appear in other fields' curve lists.

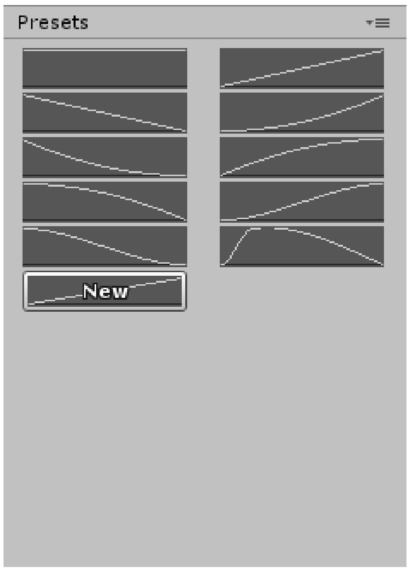


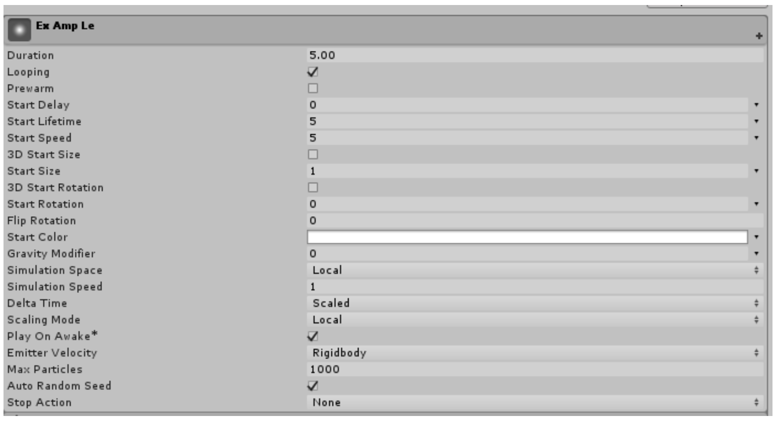
FIGURE 15 Curve presets

3.2 The Default modules

These modules are the ones that are activated by default in each Particle System. While the main module and renderer are mandatory, the Shape and Emission modules can be disabled in certain scenarios to achieve desired results. These “default” modules are the framework one builds their particle upon and as such knowing how to use them properly is essential.

3.2.1 The Main Module

We start off with the all-encompassing **Main** module, also known as the Particle System module, that has a boatload of fields that contain mostly options for the start of a particles emission.



Ex Amp Le

Duration	5.00
Looping	✓
Prewarm	☐
Start Delay	0
Start Lifetime	5
Start Speed	5
3D Start Size	☐
Start Size	1
3D Start Rotation	☐
Start Rotation	0
Flip Rotation	0
Start Color	⬜
Gravity Modifier	0
Simulation Space	Local
Simulation Speed	1
Delta Time	Scaled
Scaling Mode	Local
Play On Awake*	✓
Emitter Velocity	Rigidbody
Max Particles	1000
Auto Random Seed	✓
Stop Action	None

FIGURE 16 The Main module

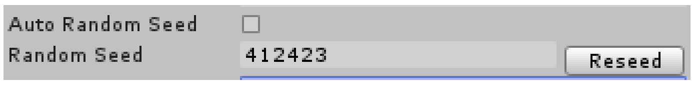
Before I hop on to the fields that affect the particles, we’ll take a look at the fields that control how the system behaves. The **Looping** field makes it so that when enabled, the system will endlessly loop the effect for as long as the object remains active and opens up the **Prewarm** option that starts the emission as if one loop had been completed beforehand. One can disable automatic emission by disabling the **Play on Awake** field and **Max Particles** designates how many particles can be active at a time from this module specifically. Finally, **Duration** handles the duration of the emission.

Now we’ll turn our eyes to the **Start** fields, these fields specify the values that the particle will have when its emitted. **Lifetime**, **Speed**, **Rotation** and **Size** control the respective properties of the particle and **Start Delay** adds a delay based on the amount to the start of the emission. **3D Start Size** and **3D Start Rotation** enables one to separate the axes and manipulate them separately, although the Z-axis won’t work for **Size** unless they have a mesh as their particle.

Start Color is a bit different, in it that it only really works if one has made the texture white from the get-go. One can still change the opacity and the color they have specified applies as a tint over the textures original color if they had it colored beforehand.

Now there are a few more fields one should be aware of that impact the overall system. First off **Gravity Modifier** applies the global gravity (found under Physics settings) value based on the modifier to the particles, it is set to zero by default which turns gravity off. **Simulation Space** determines whether one's particles move with the parent object, a custom object or the world. (Probably need to specify a bit)

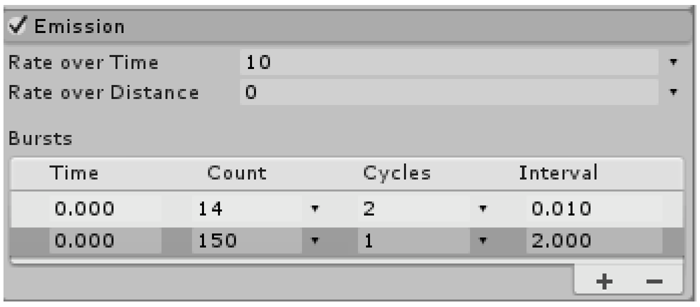
If one has used the Particle System before, they may have noticed that the effect is never the same over multiple emissions, this is because of **Auto Random Seed** which is enabled by default. If one disables this field, they can specify a random emission in the **Random Seed** field and once this is done it will play the same effect each time, because a random emission was specified as shown in figure 17. A bit confusing, I know.



Auto Random Seed	☐
Random Seed	412423

FIGURE 17 Random seed setup

3.2.2 Emission



✓ Emission

Rate over Time	10
Rate over Distance	0

Bursts

Time	Count	Cycles	Interval
0.000	14	2	0.010
0.000	150	1	2.000

FIGURE 18 The Emission module

This module, as the name implies, handles the number of particles one emits based on the parameters they have set it to. **Rate over Time** is the number of particles that is emitted over a period of one second while **Rate over Distance** is the exact same, but for one distance unit.

One can also configure the system to emit **Bursts**. These can be used in conjunction with both emission modes or as a standalone. One can add as many bursts as they want and configure them as they see fit.

3.2.3 Shape

The **Shape** field determines the shape of the systems' emission and has a wide variety to choose from. These are far too numerous to showcase here, but instead we take a look at the different fields available in each mode. Figure 19 showcases all of the available shapes as of Unity 2018.1.

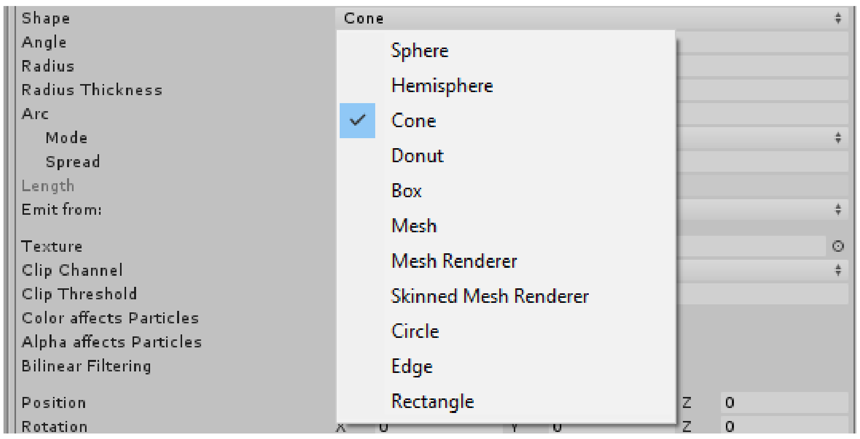


FIGURE 19 The different shapes

Many of the different shape choices have the field **Radius**. In all cases this controls the size of emission area, this can be further manipulated with **Radius Thickness**, which designates the are from where the particles emit. If this is set to zero, the particles only emit from the edge of the area.

The **Cone** shape has two unique fields in **Length** and **Angle**. **Angle** handles the shape of the cone and **Length** the length of the cone, but only when its set to emit from its **Volume**.

Donut, **Cone**, **Edge** and **Circle** share the **Arc**, **Mode** and **Spread** fields. **Arc** designates the angular area that is the shape of the emitter, while **Spread** allows one to manipulate the emission intervals around the arc. A value of 0.1 would mean that emission occurs in ten spots and a value of 0.5 means two points of emission.

Mode indicates which form of particle generation is used on the arc. It has four different options **Random**, **Loop**, **Ping-Pong** and **Burst Spread**. From last to first, **Burst Spread** is best utilized with burst emissions, as it evenly distributes emissions along the arc. **Loop** creates particles along the arc endlessly in the same order, while **Ping-Pong** works the same way, but goes in the other direction once it has completed a cycle. **Random** makes them appear from random locations over the duration around the arc.

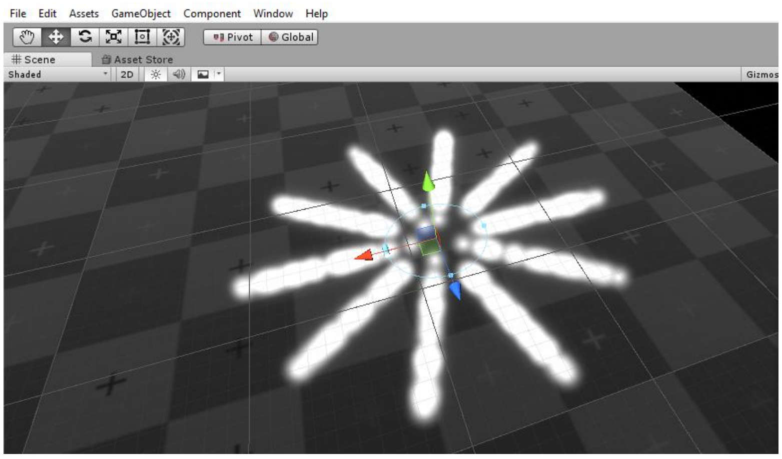


FIGURE 20 Example of burst spread in action at 0.1 spread

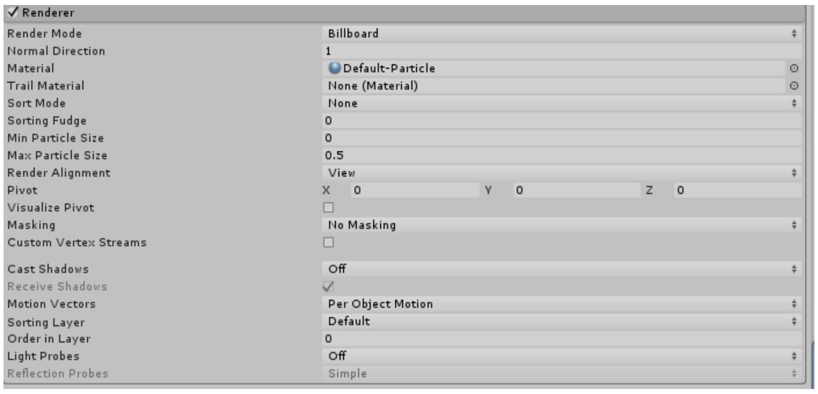
Some of the shapes have an option to choose where to **Emit From** on the shape. These options include **Edge**, **Shell** and **Volume**. With **Edge**, the emission occurs only on the edges of the vertices and while using **Shell**, the emission area is the whole surface area. **Volume** makes it so that the emission occurs also from within the shape.

Mesh shape has its own options for emissions, it can choose to emit from its vertices, edges or triangles via **Vertex**, **Edge**, **Triangles** respectively by using the **Type** field.

Along all of the above, every shape has the **Position**, **Rotation** and **Scale** fields to control the respective aspects of the emission area itself.

There are a few ways to control direction of the emitted particles, **Align to Direction** which forces the particles to align to their starting direction, **Randomize Direction** that mixes up the directions of the particles at birth, **Spherize Direction** forces them to emit outwards from the center of the gameobject and **Randomize Position** that applies randomness to the spawning positions themselves. The latter three fields can have a value of zero, where the field will have no effect at all.

3.2.4 Renderer



✓ Renderer

Render Mode	Billboard
Normal Direction	1
Material	Default-Particle
Trail Material	None (Material)
Sort Mode	None
Sorting Fudge	0
Min Particle Size	0
Max Particle Size	0.5
Render Alignment	View
Pivot	X 0 Y 0 Z 0
Visualize Pivot	☐
Masking	No Masking
Custom Vertex Streams	☐
Cast Shadows	Off
Receive Shadows	✓
Motion Vectors	Per Object Motion
Sorting Layer	Default
Order in Layer	0
Light Probes	Off
Reflection Probes	Simple

FIGURE 21 The Renderer module

The **Renderer** module contains the settings that manage how one’s particles are displayed in the game and the general drawing settings of them. Particles are flat 2D objects that align themselves to appear 3D in Unity and in the **Renderer** module one has multiple choices of **Render Mode** that decide how their billboards align themselves and by what rules do they play.

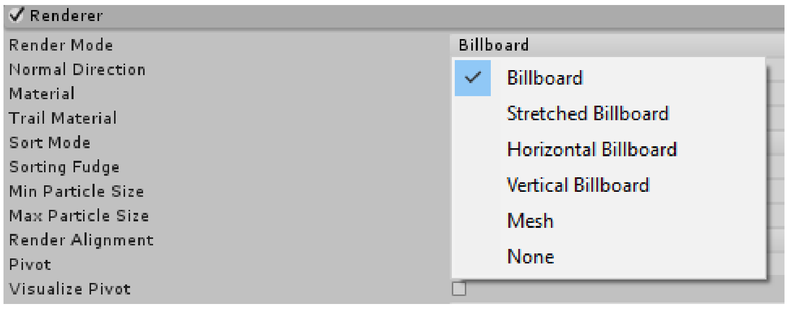


FIGURE 22 The rendering modes available to the particles

First renderer mode, **Billboard**. This is enabled by default and makes it so that the particles always face the camera. This is mostly used for things that look similar from all angles like gas and such.

Second mode is **Stretched Billboard**, which always makes the particles face the direction and the camera. This mode unlocks new options **Camera Scale**, **Velocity Scale** and **Length Scale**. **Camera Scale** stretches them according to the movement of the camera, while **Length Scale** stretches them based on the multiplier to the direction they are heading. **Velocity Scale** stretches the particles based on their speed. All of these fields can be set to zero, except **Length Scale**, as leaving that to zero makes the particle disappear.

Horizontal Billboard aligns the particles along the XZ axes, this is useful for effects that are on the ground or flat objects. **Vertical Billboard** makes it so that the particles stay upright along the XZ-axes and align themselves on the Y-axis while staying rotatable.

Mesh mode allows one to render a 3D mesh instead of a texture, for example boxes. Usually with this mode one wants to change their material, since textures are not usually a good fit with meshes.

Finally, **None**, which completely removes the rendering of the particle and leaves only the **Trail** to be rendered.

To change the direction that the **Billboard** and **Mesh** render modes face, use the **Render Alignment** field. One can either make them face the camera with **View** or align them with the world axes with **World**. **Local** aligns them with their own transform and **Facing** directly with the camera object. Finally, **Velocity** aligns them with the direction they are going.

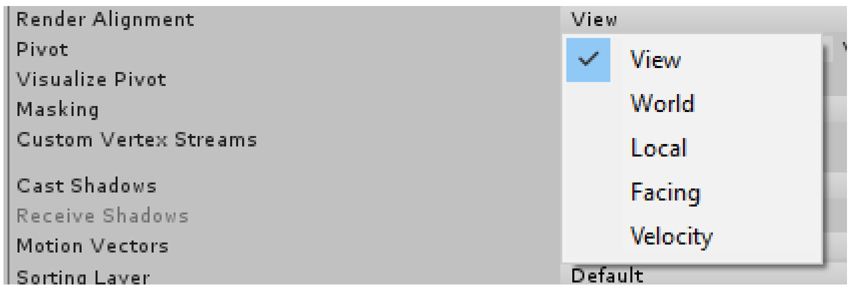


FIGURE 23 Different options of Render Alignment

Both **Material** and **Trail Material** represent the respective materials used for rendering said parts. **Sort Mode** controls the order of the rendering for the particles, **By Distance** option is measured from the camera and **Oldest in Front** and **Youngest in Front** do exactly that. With **Sorting Fudge,** one can add randomness to the sorting.

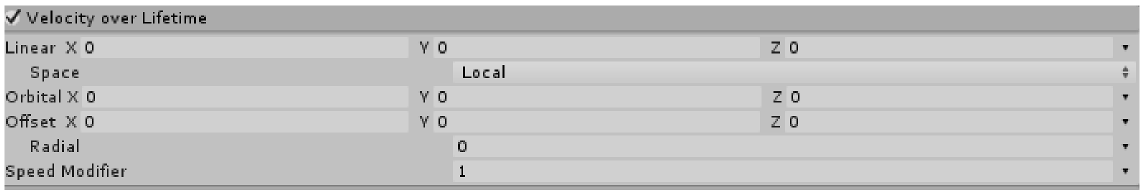
Min and **Max Particle Size** only work when the rendering mode is set to **Billboard**, but they are immensely useful fields. Whenever one zooms in on their particle, it begins to turn smaller and smaller. Setting the **Max Particle Size** field to a higher number allows one to zoom in on the particle, this is useful for when they are working with smaller particles and they want to see it up close.

Last few fields to keep an eye on are **Cast Shadows**, which allows one to control whether their particles cast a shadow and in which method or not at all. On the flipside **Receive Shadows** enables shadows to be cast on the particle itself.

3.3 Over Lifetime modules

This chapter will cover all the modules that do their magic over the lifetime of the particles. All of these modules contribute to making the particles look more lifelike in their respective ways. One does not usually use all of them but almost always at least one of them.

3.3.1 Velocity over Lifetime



✓ Velocity over Lifetime

Linear	X 0	Y 0	Z 0
Space	Local		
Orbital	X 0	Y 0	Z 0
Offset	X 0	Y 0	Z 0
Radial	0		
Speed Modifier	1		

FIGURE 24 The Velocity over Lifetime module

In this module, one manages the speed/velocity of the particles over their lifetime. There are three different methods of velocity one can add, **Linear XYZ** which applies speed along the specified axis/axes, **Orbital XYZ** which makes particles orbit around the emission point unless modified and **Radial** that makes particles emit away from the center of the object. All of these fields support the use of negative values for the reverse effect, so if one puts a -1 in the **Radial** field the particles will then emit towards the center instead of away from it.

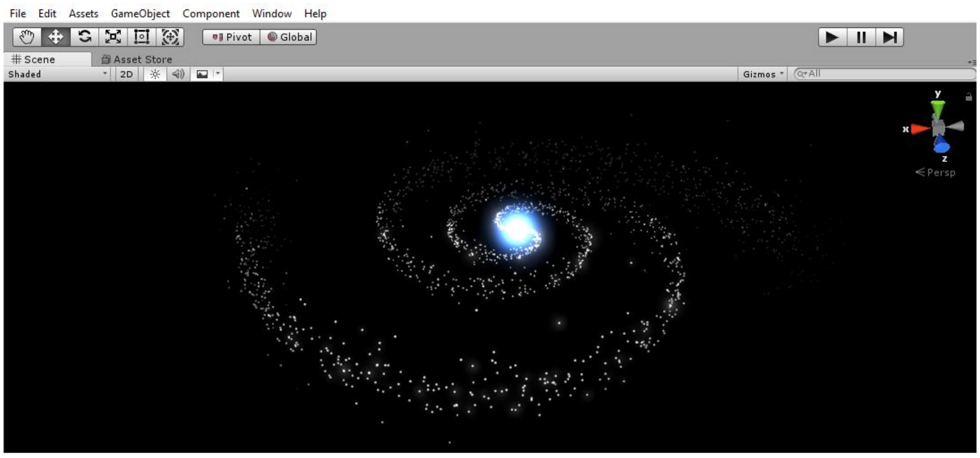
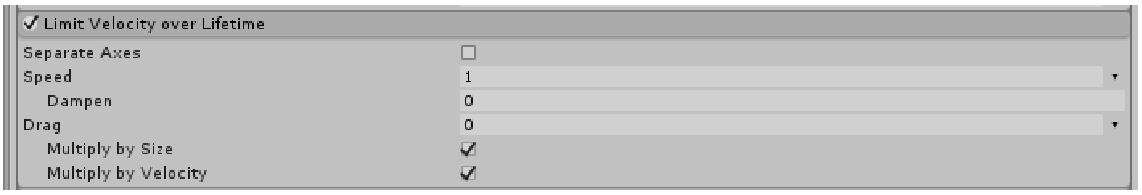


FIGURE 25 Galaxy effect made by using Orbital Velocity

There are couple of other fields that affect how the other fields operate, **Space** which decides whether one uses the local or world axes for **Linear XYZ** and **Offset XYZ** that applies an offset to the center point of **Orbital XYZ.**

Lastly **Speed Modifier**, which funnily enough I found more useful for general testing of speed overall. The way it works is it applies a global multiplier to the speed of the particles along with this modules' modifiers. So, one can set this to two and the **Start Speed** they set will also be double, making it quite efficient at testing how their particles look with different speeds.

3.3.2 Limit Velocity over Lifetime



✓ Limit Velocity over Lifetime

Separate Axes	☐
Speed	1
Dampen	0
Drag	0
Multiply by Size	✓
Multiply by Velocity	✓

FIGURE 26 The Limit Velocity over Lifetime module

This module is used to simulate particles slowing down due to resistance or whatever is affecting it. **Speed** field determines the speeds at which one starts limiting the particles, while **Dampen** is the multiplier of how much speed it loses once it goes over the limit.

Drag field assigns the amount of "resistance" the particles face over their lifetime, this combined with both **Multiply by** fields can create realistic speed reductions based on their size and speed.

Lastly, we have **Separate Axes** to modify each axis separately and once enabled, it opens up **Space** where one can specify whether it uses local axes or world axes.

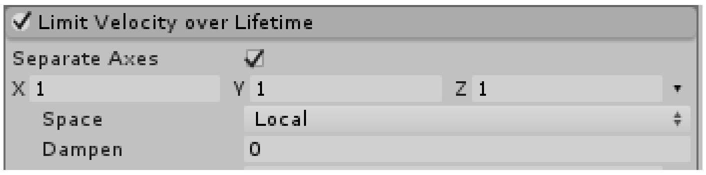

✓ Limit Velocity over Lifetime
Separate Axes ✓
X 1 Y 1 Z 1
Space Local
Dampen 0

FIGURE 27 The space field

3.3.3 Force over Lifetime

This module has only three fields, two of which are shared with **Velocity over Lifetime**, except this time they apply force to the particles instead of controlling velocity. There is a new field however, called Randomize, which only works when one has the axes set to either of the **Random between two** settings. What this does is it applies a new force direction on each frame.

Now people sometimes get confused when using this module, as this module doesn't have immediate results. This is because it applies a **Force** over lifetime and does not interact with the actual speed directly, that is the job of **Velocity over Lifetime**. The difference with the exact same settings on both modules is shown below in figure 28.

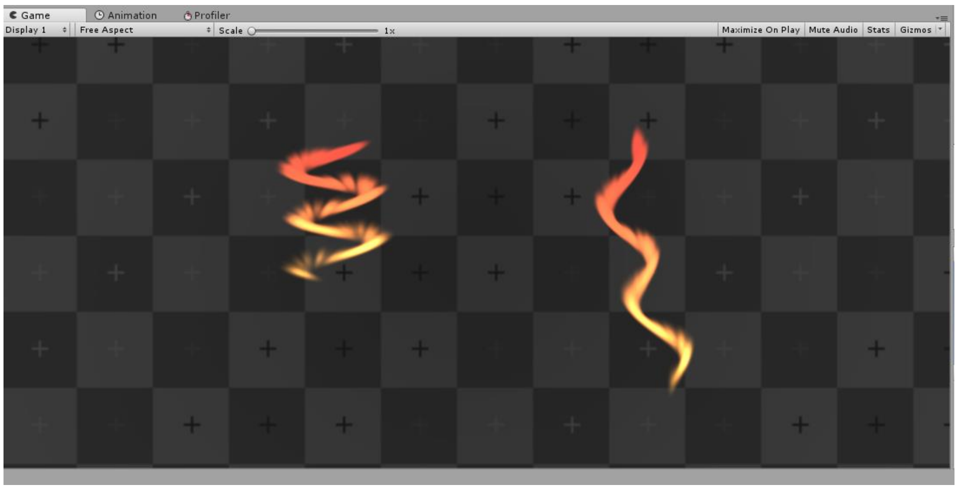
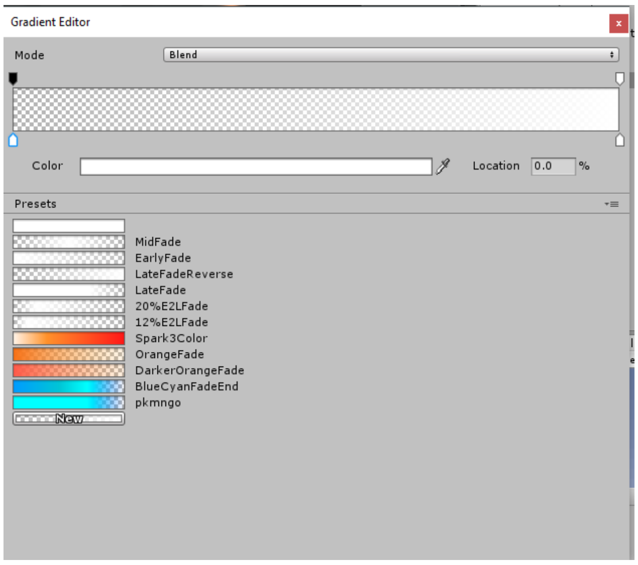


FIGURE 28 Difference between Velocity (Left) and Force (Right)

3.3.4 Color over Lifetime

Controls the color of the particle during its lifetime. It only has one field, that only has two options, **Gradient** or **Random between two Gradients**, but its uses are pretty much limitless. This module has the most effect upfront in making the particles more alive and natural. The left side of the gradient represents the beginning of the particle's lifetime and the right side the end. It is also recommended to save one's customized fades and colors for future use, trust me it saves a boat-load of time and makes testing different fade points much easier.


Gradient Editor

Mode: Blend

Color

Location: 0.0 %

Presets

MidFade
EarlyFade
LateFadeReverse
LateFade
20%E2LFade
12%E2LFade
Spark3Color
OrangeFade
DarkerOrangeFade
BlueCyanFadeEnd
pkmngo

FIGURE 29 The Gradient editor

Normally the particles just appear out of nowhere and disappear as suddenly as they appeared. With this module one can fade them in and out to make them less jarring for the players. This is achieved by making alpha channel breakpoints that represent the times when one wants the particle to start disappearing or appearing. The alpha breakpoints are the pins on top of the gradient bar and one can set as many as they like. **Location** can be used to get precise points without having to drag each pin.

The same process is used for the colors, but instead of the top row one uses the bottom row. The same rules apply, add breakpoints for different colors and decide their location to create a gradient. One can add as many breakpoints as they wish, but usually they don't want to go for more than four as shown in figure 30.

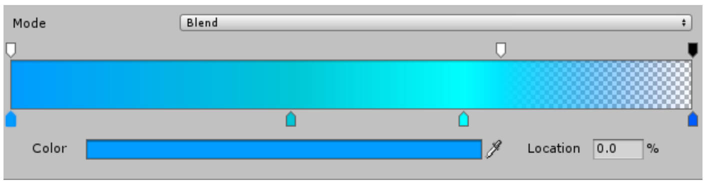


FIGURE 30 A four-point gradient that fades near the end

3.3.5 Size over Lifetime

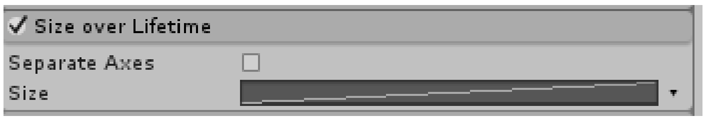


FIGURE 31 The Size over Lifetime module

Another small module that has a huge impact on the particle itself. As the name implies, this module controls the size of the particle over its lifetime and as such this is usually a **Curve**. If one wishes to control only one of axis's or all of them separately, check **Separate Axes**.

This module is something one should fiddle around with themselves to find whatever works for them. It's simple and the effects are constantly visible. One should try to think how their particle should behave, if it's a gas then it's going to expand over its lifetime, so they should use a **Curve** that goes from **low to high**, for example.

3.3.6 Rotation over Lifetime

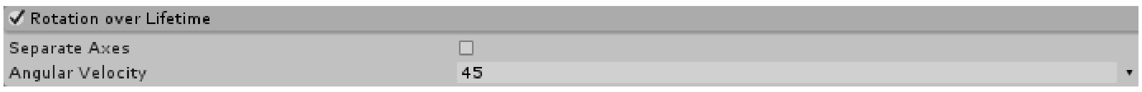


FIGURE 32 The Rotation over Lifetime module

This module can help break the monotony in one's particles by adding **Angular Velocity** to the mix. Usually one should go for a randomized value, as having a **Constant** rotation in each of one's particles can have the exact opposite effect of breaking monotony.

Another way of using this module with **Constants** is to **Separate Axis** and assign a value to each of the axes to create randomness. This also works with curves and random between two values on each axis.

3.3.7 By Speed modules

These modules work the exact same way compared to their over lifetime counterparts, except they are controlled by the speed of the particles, instead of the lifetime. The speed is set in the **Speed Range** and it only works with curves. In all three of the **By Speed** modules, the left side of the curve represents the **Speed Range** minimum value and the right side the maximum value.

The **Color, Size** and **Angular Velocity** fields work the same as before and **Separate Axis** is available to both rotation and size to create more irregular sizes and rotations.

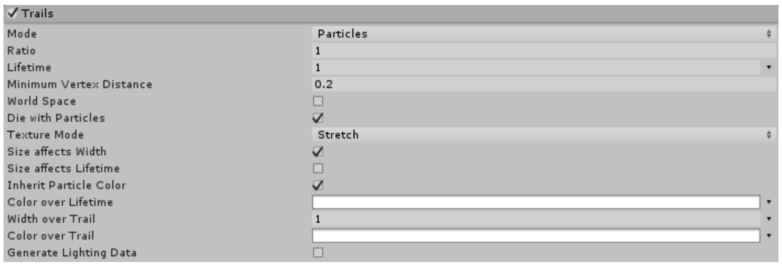
These modules can be used over the lifetime ones in the case that one wants to control the particles by their speed. For example, rising smoke slows down as it rises higher, so one increases its **Size** and turns it more transparent via **Color By Speed** creating a natural feeling smoke effect.

3.4 The other modules

Finally, in the Particle System are the modules that offer something unique and very case specific to one's particle effects. All of these modules add something that is not available anywhere else.

Custom Data was left out due to being related to custom shaders and I don't cover shaders in such a high detail to warrant a chapter on it.

3.4.1 Trails



✓ Trails	
Mode	Particles
Ratio	1
Lifetime	1
Minimum Vertex Distance	0.2
World Space	☐
Die with Particles	✓
Texture Mode	Stretch
Size affects Width	✓
Size affects Lifetime	☐
Inherit Particle Color	✓
Color over Lifetime	
Width over Trail	1
Color over Trail	
Generate Lighting Data	☐

FIGURE 33 The Trails module

In this module one adds a trail to a certain amount of their particles which can inherit values from the particle itself. This is a bread and butter module for bullets and magical effects. **Trails** can also be created by using **Rate over Distance** in the **Emission** module, but that usually ends up with having a boatload of particles, well over a thousand, so it can get heavy in certain situations.

Remember to add the **Trail material** in the **Renderer** module!

Mode field determines whether one's trails use **Particle** mode or the **Ribbon** mode. With **Particle** one leaves behind a trail that follows the parent particle and **Ribbon** makes it so that every particle is connected via a trail based on spawn order.

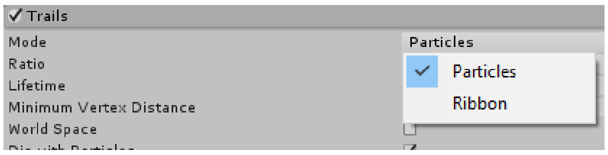


FIGURE 34 The two trail mode options

Ribbon also opens a new option, **Ribbon Count** and the way it works is based on the number put in the field, the system increments the next ribbon at that value. So, at 1 it makes a single ribbon that connects each particle, at 2 it makes two ribbons that take turns in connecting particles and with 3 it creates three separate ribbons that connect every third particle.

Final unique field for **Ribbon** is **Split Sub Emitter Ribbons**, that allows one to connect or split their ribbons created by sub emitters. This only works if one has multiple sub emitters with the same parent object though.

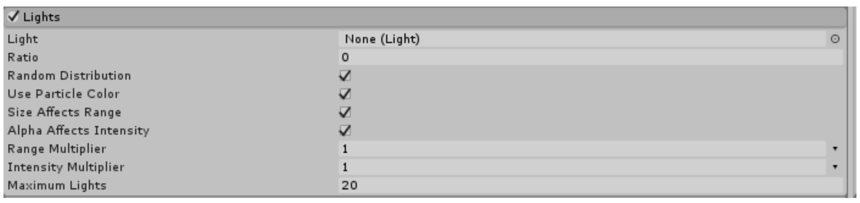
Ratio and **Lifetime** are properties only available for the **Particle** mode, **Ratio** designates how many particles should have trails added to them, the amount cannot be controlled as Unity assigns them randomly even if one sets this to the maximum value of 1. **Lifetime** is a multiplier to the lifetime of the parent particle and each vertex of the trail stays alive for this amount of time.

Last of the **Particle** mode exclusives are, **World Space** which when enabled makes the trails ignore all movement of the particles they are attached to and just fall over time and **Die with Particles**, which when enabled causes the particles to instantly die after that parent particle dies instead of staying visible for its remaining lifetime after the parent particle has died.

Sometimes the trails look a bit jagged if the particle moves around a lot, this is because of **Minimum Vertex Distance.** This setting controls the amount of distance each particle must travel before a new vertex is created for the trail. Setting this to zero creates the smoothest curves in the trail but one should play around with the value and not automatically set it to zero. Might not have a huge impact on performance, but everything counts.

Rest of the fields are used by both **Modes** and require no real explanation as they are very straight-forward, except **Texture Mode**. This field decides how the trail is displayed over the trail. The options that are most important are **Stretched** and **Tiled**. **Stretched** does exactly what it says, stretches the material over the trail and **Tiled** tiles it over trail, but that's not all, **Tiled** uses the **Tiling** setting from the material as the frequency rate.

3.4.2 Lights



✓ Lights

Light	None (Light)
Ratio	0
Random Distribution	✓
Use Particle Color	✓
Size Affects Range	✓
Alpha Affects Intensity	✓
Range Multiplier	1
Intensity Multiplier	1
Maximum Lights	20

FIGURE 35 The Lights module

Another module which adds a ton of liveliness to the particles, especially if one takes care and matches the size of the lights to the size of the particle. Combine this with inheriting the color of the particle and they will look much more believable.

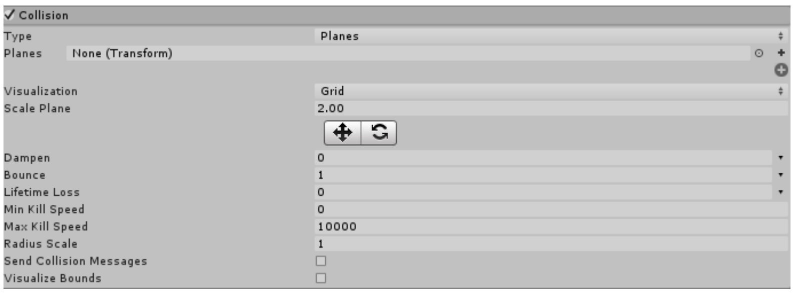
As with the module before this one (Trails), the **Lights** module also applies its effects on a random number of particles which one has no control over. Adding boatloads of real-time lights can get heavy on the system, especially if one hasn't specified which objects in their scene reflects lights or casts shadows. After one has added their lights to the particles, I highly recommend they monitor their performance.

First of we'll start with the most important field in this module, **Maximum Lights**. With this one can regulate how many lights are allowed to be spawned simultaneously and prevent any crashes or major lag from happening because of it.

For the **Lights** module one has to make a prefab light object and put it in the **Light** field, the prefab will retain all of its properties (Range, Intensity, Color) and one can modify them from the module via **Range Multiplier**, **Size Affects Range**, **Alpha Affects Intensity**.

For the ratio of the lights appearing in one's particles use **Ratio**, the number used is a representation of the number of particles that receive a light. **Random Distribution** is used to determine whether particles receive lights randomly or at a set pace. Whether or not one has random distribution enabled, **Ratio** still controls the frequency of lights appearing.

3.4.3 Collision

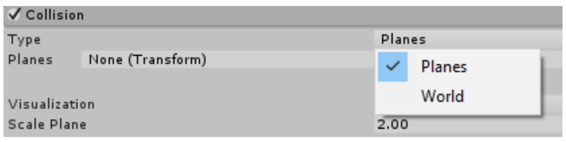


✓ Collision

Type	Planes
Planes	None (Transform)
Visualization	Grid
Scale Plane	2.00
Dampen	0
Bounce	1
Lifetime Loss	0
Min Kill Speed	0
Max Kill Speed	10000
Radius Scale	1
Send Collision Messages	☐
Visualize Bounds	☐

FIGURE 36 The Collision module

As the name implies, this module deals with collisions of the particles with the world. This can be set to a custom plane via the **Type** field or the whole world. If world is chosen, one can set it to either a 2D world setup or a 3D one.



✓ Collision

Type	Planes
Planes	None (Transform)
Visualization	
Scale Plane	2.00

FIGURE 37 The two collision types

Now when one thinks of collisions, they would usually associate it with something ricocheting or bouncing, which would be correct, although one can also set it so that the particle is killed on impact. This is governed by the **Min Kill Speed** and **Max Kill Speed** where one can set the speeds wherein upon impact the particle gets destroyed. **Lifetime loss** on the other hand reduces the particles' lifetime itself by the amount specified in the field on each collision.

If one decides to not to kill the particles on impact, they should turn their eyes to both **Dampen** and **Bounce** fields as they control the behavior of colliding particles on collisions. Both of them are fraction multipliers for the speed of the particle, **Bounce** is the amount of speed the particle with-holds after bounces and **Dampen** is the multiplier for the speed loss per impact.

Three more fields are shared by both modes and first off **Send Collision Messages,** which is an absolute must if one plans on following particle collisions through code via OnParticleCollision. Secondly **Radius Scale**, with this one can adjust the size of the collider in each particle so that they aren't completely out of touch with the actual size of the particle. And lastly, the **Visualize Bounds** checkbox gives one access to a visual of the actual size of the collider, so they can better adjust it to match the particle.

Now there are other differences in the two options of the **Type** field, when one has the planes option selected, they can make a list out of the planes they want to collide with and they can also visualize it in wireframe mode by using the **Visualization** field.

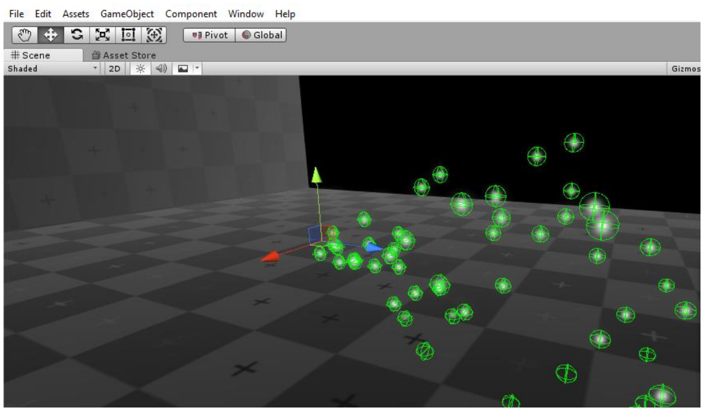
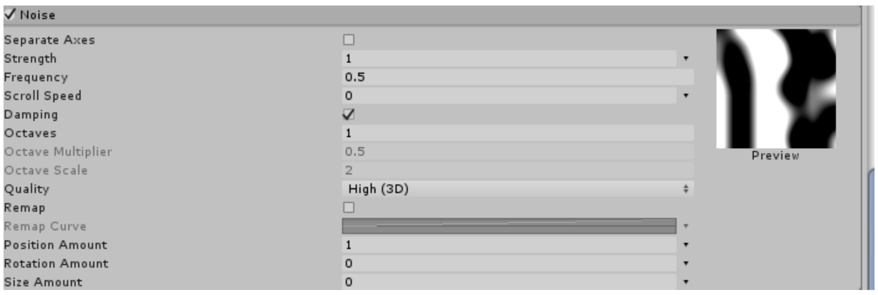


FIGURE 38 The effect of the Visualize Bounds checkbox

The world setting has **Collision Quality** that when set to lower settings can cause some collisions to not be registered at all, but in return it improves performance. One can also disable certain layers from being eligible to be collided with by using the **Collides With** option. Controlling what the particles can and cannot collide with is extremely useful and should be used almost always.

3.4.4 Noise



✓ Noise

Separate Axes	☐
Strength	1
Frequency	0.5
Scroll Speed	0
Damping	✓
Octaves	1
Octave Multiplier	0.5
Octave Scale	2
Quality	High (3D)
Remap	☐
Remap Curve	
Position Amount	1
Rotation Amount	0
Size Amount	0

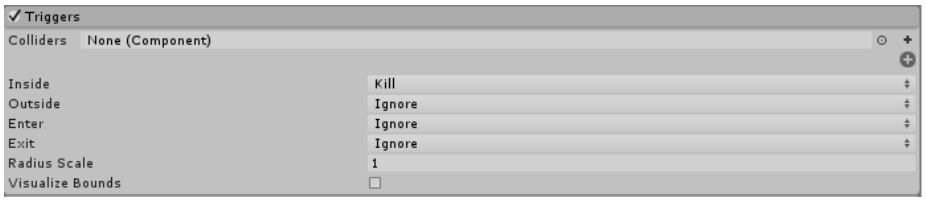
FIGURE 39 The Noise module

The absolute randomness module. With this module one can add crazy and erratic movement to the particles but it's also heavy on performance if they go with higher quality noise. While it is performance heavy though, it still creates the most unique movement patterns in the Particle System.

Strength controls the magnitude of the noise on each particle while **Frequency** regulates the changes in movement that the particles have. **Scroll Speed** on the other hand increases or decreases the unpredictability of the noise based on the value it has. With these three fields one controls the most important parts of the Noise module and the noise itself, these should be toyed around with a lot to find just the right amount of randomness one wants their effect to have.

Other notable fields include **Separate Axes**, **Damping, Octaves, Quality** and "**Amount**". Separate axis is the same as before, control each axis independently, **Damping** on the other hand works differently from before, as now it specifies whether or not **Strength** is proportional to the **Frequency** value. This allows the system to be scaled while keeping the noise similar. The **Amount** fields are multipliers to control how much each of them are affected. **Octave** designates the number of layers in their noise, so the higher it is the better the noise is, but at a cost to performance. **Quality** is pretty much the exact opposite of **Octave**, as the lower one gets (3D-1D) the more stale their noise gets but increases performance as a result.

3.4.5 Triggers


✓ Triggers

Colliders	None (Component)
Inside	Kill
Outside	Ignore
Enter	Ignore
Exit	Ignore
Radius Scale	1
Visualize Bounds	☐

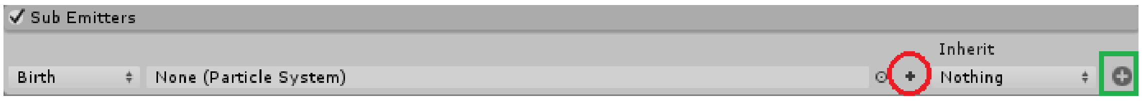
FIGURE 40 The Triggers module

This module, when enabled, gives one's particles the capability to send Callbacks whenever they collide or interact with other colliders set to be Triggers. But to do so, first one must set with which colliders they can interact with by using the **Colliders** list. After they have done so, they can start to set the actions to take for each event, while keeping in mind that these apply to all the colliders listed.

The module has some fields already encountered in the Collision module and they work exactly the same here, those two being **Visualize Bounds** and **Radius Scale**. For the new ones we have **Inside**, **Outside**, **Enter** and **Exit**, these all represent different trigger events and are self-explanatory.

What is the most important part of this module are the options for what to do whenever each event occurs. **Callback** sends a message that can be intercepted in code and be programmed to do all sorts of behavior, like changing color or triggering a sub emission. This will be covered more in chapter 4.2. **Ignore** sets the event to be ignored whenever it occurs for that systems' particles and **Kill** destroys the particles once the event is called.

3.4.6 Sub Emitters


✓ Sub Emitters

Birth | None (Particle System) | Inherit: Nothing

FIGURE 41 The Sub Emitters module

Sub Emitters are particle systems in one's system that emit when certain events occur in the main system. This can be used to create fireworks or ripples made by rain hitting the ground and other effects that have multiple phases. A thing to keep in mind however, is that one really does have systems in their systems so keeping an eye on the particle count is recommended as it can get massive quite rapidly.

All of the Sub Emitter emissions happen in the current position of the particle and those emission events can be set to occur upon either **Birth, Collision, Death, Trigger** or **Manual**, now all of these except one is straightforward and that one is **Manual**. If one sets their Sub Emitter to manual, they can call it to happen through code which is covered in chapter 4.2.

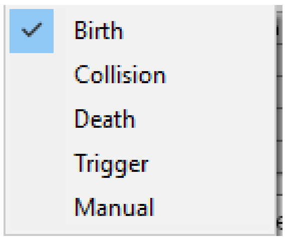


FIGURE 42 The different trigger modes

Out of those five only **Birth** is not restricted to using Burst Emissions only. For the other four one has to use burst emissions, or they will not emit anything, the Sub Emitter is automatically set to use Bursts if one generated a new system by using the plus sign next to the **System** object slot.

In the **Inherit** options one can set all the things they want their Sub Emitter particles to inherit from the parent particles. These include Size, Color, Lifetime and Rotation. Velocity can also be inherited by turning on the module **Inherit Velocity** on the Sub Emitter.

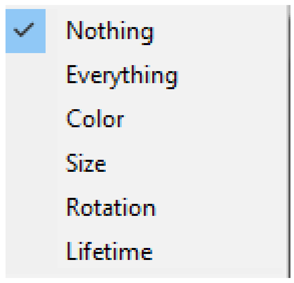
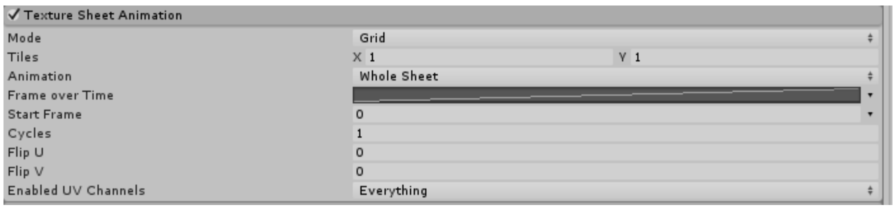


FIGURE 43 The options for inheritance for Sub Emitters

3.4.7 Texture Sheet Animation



✓ Texture Sheet Animation

Mode	Grid
Tiles	X 1 Y 1
Animation	Whole Sheet
Frame over Time	
Start Frame	0
Cycles	1
Flip U	0
Flip V	0
Enabled UV Channels	Everything

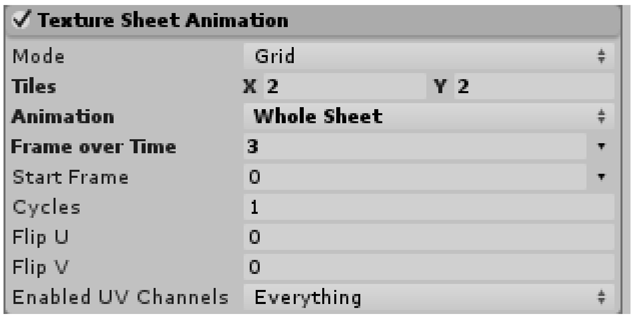
FIGURE 44 The Texture Sheet Animation module

With this module one can create flickering flames and other effects that give the impression of movement. To make this happen, one needs a material that has a **Texture Sheet** attached to it, we covered what these are in the Materials chapter. Another way to make use of this module is by selecting the other **Mode** property, Sprites. This mode requires one to use sprites that are on the same texture, so make sure to pack them or make them into an atlas.

Couple of unique options to the Grid **Mode** include **Tiles**, which is used to map the sheet so if one has an 8x8 texture sheet they map the X and Y accordingly to 8 and 8 respectively, **Animation** which designates whether it animates the whole sheet or just a single row**,** and finally **Random Row** and **Row** which decide the row in the sheet used for the animation. **Row** can only be selected if **Random Row** is disabled.

Rest of the options are shared by both modes and starting off with **Frame Over Time** which uses a curve to display the frames over time. **Start Frame** can be set to a different position if one wants a specific frame to start the animation. This setting can be set to Random between two Constants for a more natural looking emission. **Cycles** determines how many times the sheet is looped over and finally **Flip U & V** that causes a certain number of particles to be emitted mirrored.

Now this is not the only use of this module, as discussed before in the Materials section one can make multiple textures on a sheet that don't animate to save some draw calls. If one has a 2x2 sheet with four different textures on them, they can choose which one of those to emit by using **Start Frame** and **Cycles**. Map the sheet normally using the **Tiles** field and set the **Start Frame** to the desired texture and finally the **Cycles** to zero. This way one can emit four different textures from the same material. Example settings are displayed in figure 45.



✓ **Texture Sheet Animation**	
Mode	Grid
Tiles	**X 2** **Y 2**
Animation	**Whole Sheet**
Frame over Time	**3**
Start Frame	0
Cycles	1
Flip U	0
Flip V	0
Enabled UV Channels	Everything

FIGURE 45 Multiple material settings example

3.4.8 External Forces

✓ External Forces	
Multiplier	1

FIGURE 46 The External Forces module

With this module one can use Wind Zones with their particles, it is an okay tool for a specific purpose that is not usually needed at all. I personally couldn't get this to work in the way I liked even with tinkering the Wind Zone object itself and not just the **Multiplier** field in the module.

That being said, in Unity 2018.3b the External Forces module got some new tools and looks a lot more useful than before. Many new options for the module with the ParticleSystemForceFields they added.

4 GOING BEYOND THE SYSTEM

So far, we've covered the basics of effects and what kind of parts make up the Particle System. Now we'll look at things to add to one's particles that will either make life easier or open up options that they didn't have before. Lastly, a quick look at optimization and the endless headaches that come with it coupled with a few examples from different games of things done right and some worse ones.

4.1 UI Extensions

For some reasons this great package is not a part of Unity on its own AND I hadn't heard about this until I came across it in a video a month ago. Boy what a gamechanger for particles in the UI and 2D games in general! One does not have to get absolutely everything from the package, just the parts that have something to do with particles.

What the extensions enables one to do is it allows them to align their particles to the **UI Canvas** simply and efficiently. This helps with making UI effects and removes the need for manually aligning stuff in the scene.

Now a word of warning though, as it is not integrated into Unity, there is absolutely zero guarantee that it won't cause problems. But the upsides are damn good, and I would still recommend getting it regardless of the potential problems it brings.

4.2 Coding

While the most common use for coding with particles is most likely spawning the object itself with instantiate, the Particle Systems can also be affected through code. Every module that the system has can be accessed and modified through code. One doesn't have to code anything to make particles, but it is also integral to both the **Trigger** and **Sub Emitter** modules and opens options that one simply didn't have through the editor.

The big takeaways and most used ones from experience are **Emit**, **TriggerSubEmitter** and anything to do with the state of the system like **Play**, **Pause**, **Stop** and their Boolean checks **isPlaying**, **isPaused**, **isEmitting.** There are many other options that one should take a look at in the Unity manual, if they wish to manipulate systems through code. **Emit** is the one I've used the most as it simply emits the number of particles in that instant. This was showcased in the Unity tutorial for a ParticleDecalDrawer and the whole series was an excellent display of coding with the Particle System. The Boolean checks are useful for determining the state of one's system and for setting up a failsafe.

If one sets the **Sub Emitter** mode to **Manual,** then they can call the emission to happen with **TriggerSubEmitter**. Activating the sub emitters through code is pretty much the only way to get the emission timed with things as the four other ways to trigger a sub emitter are very rigid and specific. One could try and time the emissions through **Start Delay** in the **Main** module but even then, they run into the issue of it not being modular at all.

If any of the fields in the **Triggers** modules are set to callback, they can be intercepted via code and the properties can be changed at the time of the trigger. Within this call one can change particle properties or emit a sub emission and change its color afterwards, the possibilities are endless.

4.3 Animations

Animations are also available for use with particles. They work in the same manner as with any other object. One selects the object they want to animate, then create the animation. After that they choose the attribute they want to animate from the particle and record the animation by using the keyframes to adjust the desired properties.

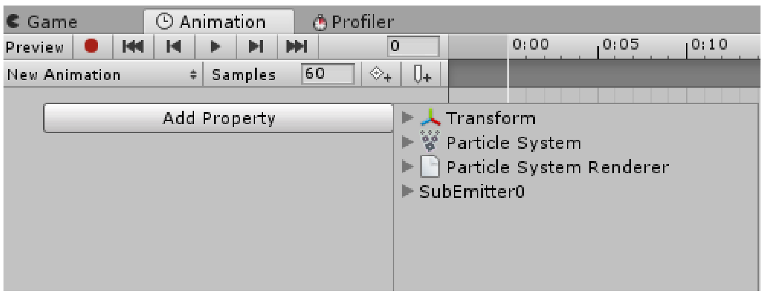
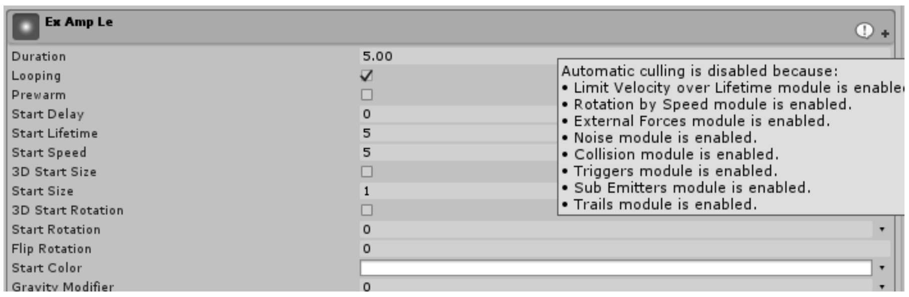


FIGURE 47 Animation window after creating the animation

With animations one can make moving particles much easier, such as an animation where particles move along the edge of a card or a controlled movement across an area or a screen. This could be done through code, but why make it so difficult when one can simply move the object in their editor and set the keyframes accordingly.

4.4 Optimization

One of the most overlooked part of the Particle System is the small icon shown in figure 48. This icon indicates that the particles are no longer eligible to be culled and as such one should always keep an eye out for it since it can have huge impact on their performance.



Ex Amp Le

Duration	5.00
Looping	✓
Prewarm	☐
Start Delay	0
Start Lifetime	5
Start Speed	5
3D Start Size	☐
Start Size	1
3D Start Rotation	☐
Start Rotation	0
Flip Rotation	0
Start Color	
Gravity Modifier	0

Automatic culling is disabled because:
- Limit Velocity over Lifetime module is enable
- Rotation by Speed module is enabled.
- External Forces module is enabled.
- Noise module is enabled.
- Collision module is enabled.
- Triggers module is enabled.
- Sub Emitters module is enabled.
- Trails module is enabled.

FIGURE 48 Culling disabled icon is hidden in the top-right corner

Another fairly obvious one is the number of particles emitted, but a more obscure one that one doesn't really realize right off the bat is how long the particle lives. When one uses the **Color over Lifetime** to fade something out, especially if it is faded out early, it still causes draw calls in the background. One can visualize this by using the **Draw Outlines** gizmo, which most people have turned off when working with particles as it gets in the way of seeing the effect.

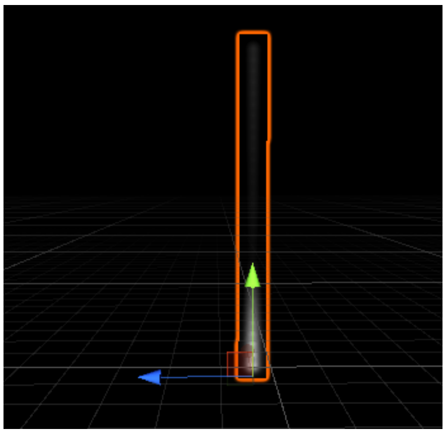
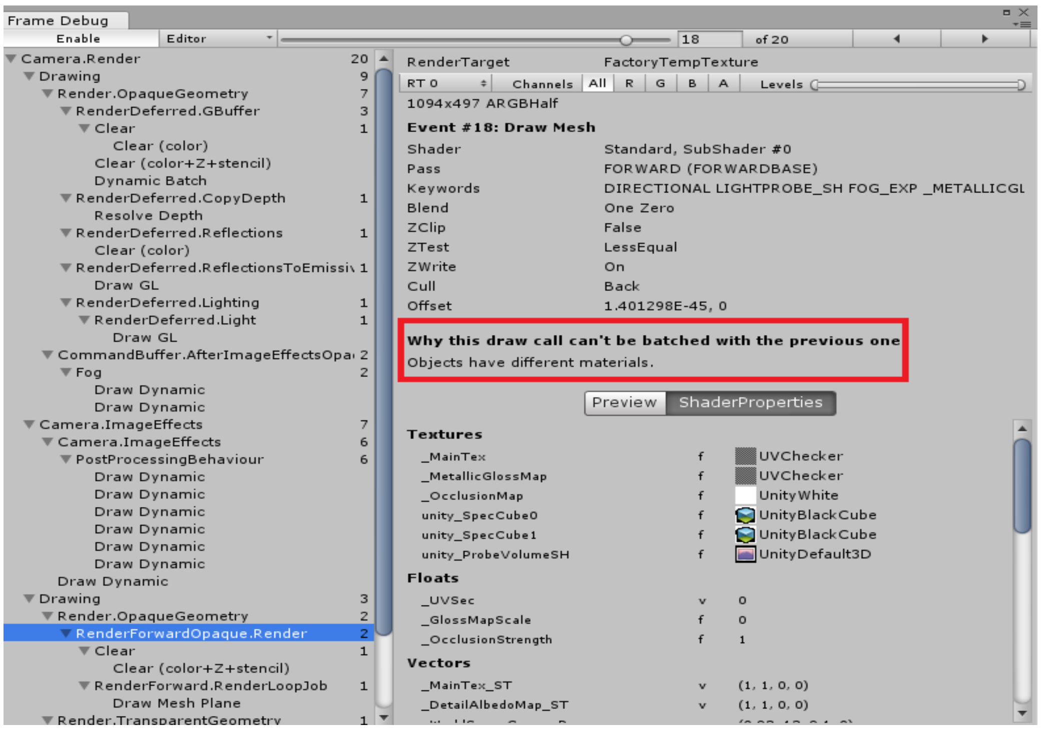


FIGURE 49 Draw Outlines gizmo enabled on faded out particles

Instantiated system should be destroyed as soon as possible, but also take into account the lifetime of the particles so that they do not simply vanish before completing their effects. **Meshes** along with **Lights** are a deadly combination together. **Meshes** alone can be tricky if one emits complex meshes in multiples but combined with realtime lights they can cause major issues. Make sure to check that **Cast Shadows** and **Receive Shadows** are not enabled on the mesh unless one specifically wishes to do so.

A lot of the above problems can be diagnosed in the **Frame Debugger**. This tool shows what is being drawn each frame and gives one reasons as to why it could not **Batch** the draw call. A good way to reduce draw calls is to use multiple textures in a single material as described in both the **Materials** and **Texture Sheet Animation** chapters.



```
▼ Camera.Render                                20
  ▼ Drawing                                     9
    ▼ Render.OpaqueGeometry                     7
      ▼ RenderDeferred.GBuffer                  3
        ▼ Clear                                 1
            Clear (color)
          Clear (color+Z+stencil)
          Dynamic Batch
      ▼ RenderDeferred.CopyDepth                1
          Resolve Depth
      ▼ RenderDeferred.Reflections              1
          Clear (color)
      ▼ RenderDeferred.ReflectionsToEmissiv     1
          Draw GL
      ▼ RenderDeferred.Lighting                 1
        ▼ RenderDeferred.Light                  1
            Draw GL
    ▼ CommandBuffer.AfterImageEffectsOpa        2
      ▼ Fog                                     2
          Draw Dynamic
          Draw Dynamic
  ▼ Camera.ImageEffects                         7
    ▼ Camera.ImageEffects                       6
      ▼ PostProcessingBehaviour                 6
          Draw Dynamic
          Draw Dynamic
          Draw Dynamic
          Draw Dynamic
          Draw Dynamic
          Draw Dynamic
    Draw Dynamic
  ▼ Drawing                                     3
    ▼ Render.OpaqueGeometry                     2
      ▼ RenderForwardOpaque.Render              2
        ▼ Clear                                 1
            Clear (color+Z+stencil)
        ▼ RenderForward.RenderLoopJob           1
            Draw Mesh Plane
    ▼ Render.TransparentGeometry                1
```

```
RenderTarget              FactoryTempTexture
RT 0        Channels  All  R  G  B  A    Levels
1094x497 ARGBHalf
Event #18: Draw Mesh
Shader                    Standard, SubShader #0
Pass                      FORWARD (FORWARDBASE)
Keywords                  DIRECTIONAL LIGHTPROBE_SH FOG_EXP _METALLICGL
Blend                     One Zero
ZClip                     False
ZTest                     LessEqual
ZWrite                    On
Cull                      Back
Offset                    1.401298E-45, 0

Why this draw call can't be batched with the previous one
Objects have different materials.

Preview   ShaderProperties

Textures
  _MainTex                      f   UVChecker
  _MetallicGlossMap             f   UVChecker
  _OcclusionMap                 f   UnityWhite
  unity_SpecCube0               f   UnityBlackCube
  unity_SpecCube1               f   UnityBlackCube
  unity_ProbeVolumeSH           f   UnityDefault3D
Floats
  _UVSec                        v   0
  _GlossMapScale                f   0
  _OcclusionStrength            f   1
Vectors
  _MainTex_ST                   v   (1, 1, 0, 0)
  _DetailAlbedoMap_ST           v   (1, 1, 0, 0)
```

FIGURE 50 Frame Debugger window

Most of the optimization is done by reducing the amount of each effect to the minimum where it still looks good. For example, limiting the lights emitted in the Particle System and the Collisions that occur might seem like small things, but every small bit of it counts and if it is in one's power to do, they should do it every time.

Lastly, keep in mind how many times the system can occur within the game. 1500 particles are no big deal if that's the only system active at that time, but if one has ten of those come out and emit at the same time they will run into problems.

4.4.1 Good optimization

Hearthstone is a great example of particles done right, as in the game one almost always has only one effect playing at a single point in time. This is mostly because of the turn-based nature of the game, but it allows for some heavier effects to be used and the developers at Blizzard have definitely taken advantage of this. It is also the reason the game runs smoothly on mobile.

Pokémon GO also has its particles done very well. Most of the effects when in the world map/overworld are very simple and don't occur many times in most areas. Some places have more than 5 stops in one area and it can cause lag on older phones, but outside of fringe cases the game runs smoothly.

Also, the effects have been tuned down to a point where they still clearly indicate something, but don't feel lackluster or out of place.

4.4.2 Bad optimization

These two examples may not be particles at all, but the same principles apply to particles and these effects alike. They showcase scenarios in which something was most likely overlooked that caused people with lower spec computers a very frustrating experience in an otherwise fantastic game.

Monster Hunter World has two instances where performance takes a nose dive due to either failing to cull effects or just simply too much going on in the game. All of this is pure speculation though, as people with very similar rigs have reported vastly varying performance in these instances.

Teostra is another monster in MHW that has caused people some performance drops with its ground exploding move. The explosion effects look fantastic, but absolutely tanked the framerate to low enough values that one might as well be looking at a PowerPoint presentation.

After testing this move a couple of this while both looking at the whole effect and turning away from it, the results were the same. This suggests a problem with the culling of the effect.

Now I cannot be absolutely certain that this effects' FPS drops are because it does not have culling, but it's something that often gets overlooked in Particle Systems and as such stands as a valid example of when it really goes wrong.

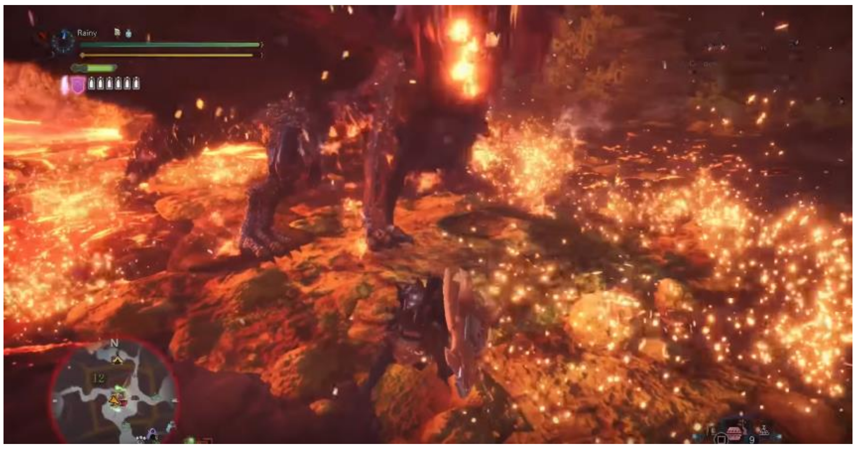


FIGURE 51 Teostra attack (Capcom, 2018)

In Path of Exile, the player character can spew out hundreds, if not thousands of objects/particles in a matter of seconds. This is mostly because the game is an ARPG and as such the player characters can reach massive attacks and casts per second. There are also items in the game that make this issue worse by automatically casting multiple spells on hit.

The sheer amount of clutter on screen can cause even the most powerful computer to lose FPS and reduce enjoyment in gameplay or outright shut people with mid-tier computers from playing these builds.

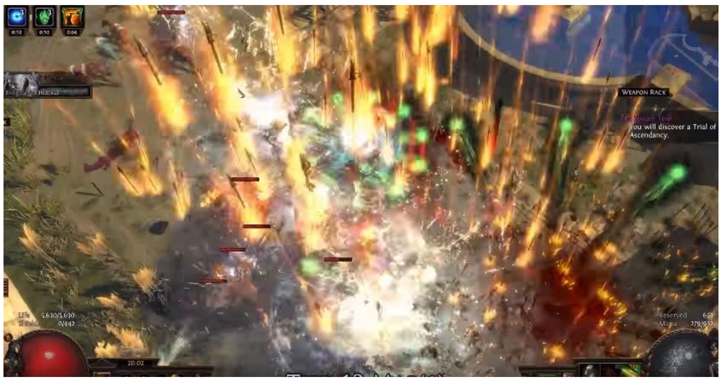


FIGURE 52 Path of Exile effects in action (Grinding Gear Games, 2018)

5 PRACTICAL EXAMPLES

5.1 Breaking it down

"Working on replicating styles you enjoy and find interesting is an excellent way to begin learning how they are created and deconstructing why decisions are made the way they are stylistically." (Matt Schwartz 2017, cited 27.11.2018)

YouTube has tens if not hundreds of guides on how to create specific effects. Some of them even come with a demonstration of how to make the texture required. Now as YouTube is so plentiful with guides on how to make a diverse array of effects, I'm not going to make cookbook guides on many, but instead show how one can break them down and replicate them to a point. In this chapter we take a look at an array of effects and what they are comprised of and how a developer might start constructing them.

Keep in mind that textures do not matter, the point is to pin down the different modules and settings used in the creation of each effect or the ones the developer could use in the case that Unity was not used as the engine.

5.1.1 Hearthstone

First, let's clear up a few things. Hearthstone is made by Blizzard Entertainment and their art team is out of this world. Not only that, but they use a wide range of shader magic and animations in their effects. As such, it is hard to say with 100% certainty which one is which, but we'll do our best regardless of the fact.

Hearthstone has hundreds of different effects to recreate so I'd recommend every developer interested in VFX or particles to take a look and try to recreate some of the effects.

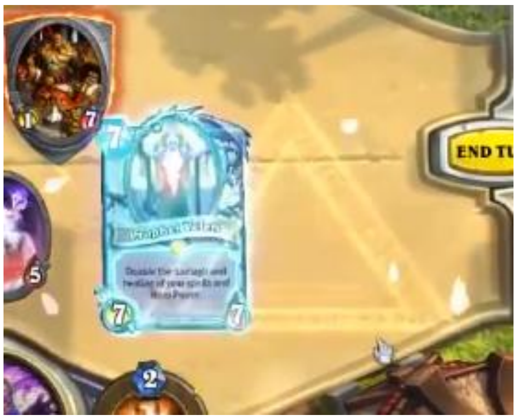


FIGURE 53 Start of Velen's effect (Blizzard Entertainment, 2016)

As the first example, we have Prophet Velen and his animation/effect when he enters the playfield. In figure 53, we see the moment right after he has been “released” or played. Now the first thing one sees are some sort of petals and a triangle. They aren’t aligned with the card, but rather with the point one drops him off on. This suggests that the effects’ **Simulation Space** is set to world as the emission point does not move with the card. The petals are random emission from the point of origin, while the triangle has a distinct texture.

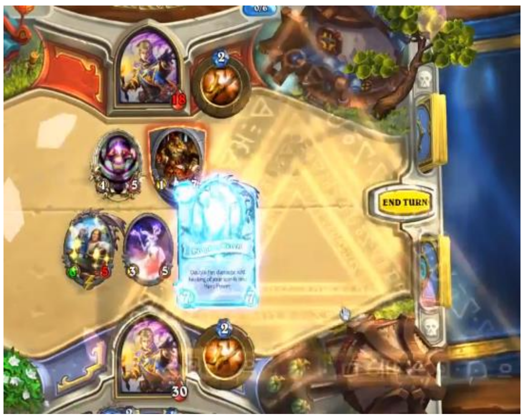


FIGURE 54 Velen a few moments later (Blizzard Entertainment, 2016)

A few moments after the emission has begun, we see that the petals have started to fade out and there are three triangles now. After seeing the effect multiple times, the timing of the triangles remained consistent which points to **Burst** emission being used at a certain interval. Also, both the triangles and petals are either being enlarged via **Size over Lifetime** or simply moved towards the camera. At this point we can also confirm the rendering mode to be either **Billboard** or **Horizontal Billboard**.

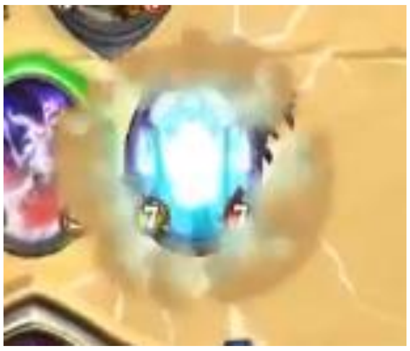


FIGURE 55 Upon impact (Blizzard Entertainment, 2016)

At this point the previous effects have disappeared and we have a **Light** and a puff of smoke or dust on the screen. The dust and the cracks in the background are likely to have been done by utilizing **Sub Emitters** set to the **Collision** option. The dust particles are emitted from the rims of the object, so a **Mesh** is used for **Shape** with the **Type** set to **Edge**. Or the shape is a **Circle** with **Random** spread. Also, the dust particles have random sizes via **Random between two Constants**.

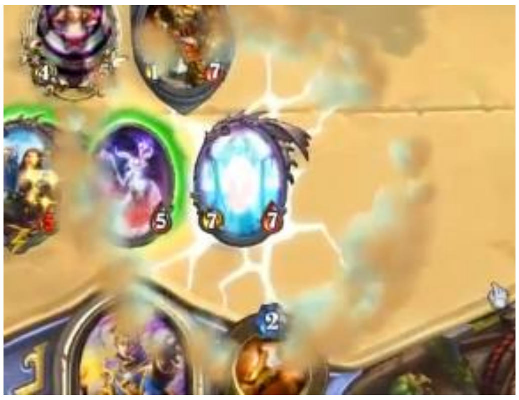


FIGURE 56 After the impact (Blizzard Entertainment, 2016)

Now we can see more clearly that the **Light** isn't hitting anything else but the smoke and that there is a fissure on the ground. The light not affecting anything else is done by using a **Culling Mask** and the fissure is a system, that has no **Shape** and emits one particle with its **Start Speed** set to zero. The dust then dissipates and disappears via **Color over Lifetime** while also seeming to slow down before completely vanishing (**Velocity over Lifetime**).

5.1.2 Pokémon GO

Pokémon GO is a game that most people are familiar with and as it is made with Unity, those two qualities make it a perfect candidate for examples. Also, the game contains a variety of effects that aren't quite as complex as Hearthstone's effects, making it a lot more beginner friendly.

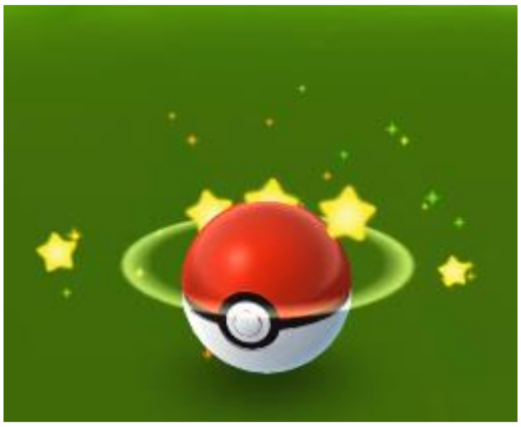


FIGURE 57 Catch effect from Pokémon GO (Niantic, 2016)

The Pokémon catching effect is a simple effect only containing three different parts. The ring, made with rendering mode **Horizontal Billboard** and a single emission, the stars and glitter which are emitted in **Billboard** mode and use a single **Burst** emission. The glitter has a **Gradient** as their color, while the others are static in color. Also, both the stars and the glitter are emitted from a **Cone** shape, this is evident from the way they spread outwards from the center and upward. To time this all at once, two of these effects are **Sub Emitters** and emit at the birth of the parent particles.

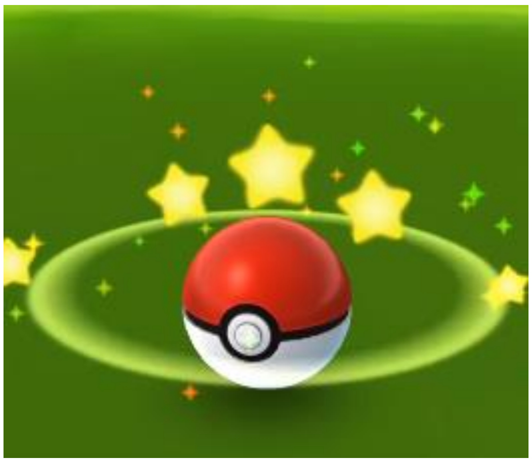


FIGURE 58 Later in the catch effect (Niantic, 2016)

At the start of the effect the stars were tiny and now as we are closer to the end, they have grown much bigger and then they suddenly shrink and disappear. This done via **Size over Lifetime** with a curve that peaks late and drops off fast. The stars also rotate, so **Rotation over Lifetime** is also at play. **Size over Lifetime** is also used to expand the ring around the ball and along with the glitter, it fades out using **Color over Lifetime**.

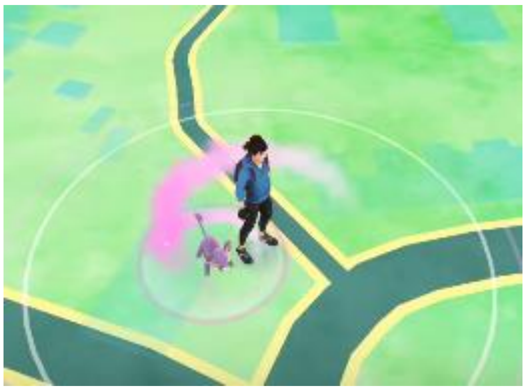


FIGURE 59 The incense effect and pulsing ring from Pokémon GO (Niantic, 2016)

When one selects a Pokémon in the overworld map, a circle appears fading in via **Color over Lifetime** and it closes in on the Pokémon with **Size over Lifetime**. The player also emits a "pulse" at certain intervals which is of a similar type as the catching effect ring, it expands outwards using **Size over Lifetime** and fades out. Both also use **Horizontal Billboard** rendering mode. The incense effect is most likely a **Rate over Distance** emission type system that utilizes **Velocity over Lifetime** and the field **Orbital Velocity** to create the circular motion.

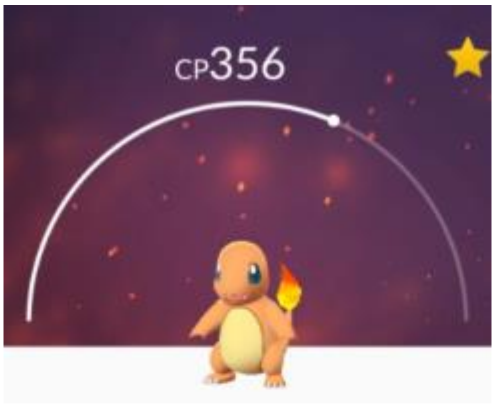


FIGURE 60 Pokémon GO fire type background (Niantic, 2016)

All Pokémon have a background in Pokémon GO, but not all of them have particles. Leaf types get swirling leaves and fire types get glowing orbs that pass the screen in the background. There isn't much to say about them other than that a lot of them use **Noise**, some also have **Lights**. (RE-THINK)

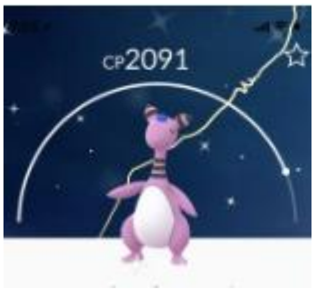


FIGURE 61 The electric type effect variation (Niantic 2016)

A lot of the background effects in Pokémon GO have some sort of variety effect to not make them look stale, in the electric type background one can see a slowly creeping lightning bolt every now and then. Most likely the bolt was made using both the **Noise** and **Trails** modules. Also, the rendering mode has been set to **None** as there is no parent particle in sight.

5.1.3 Path of Exile

Path of Exile is a game with hundreds of VFX to choose from, but it is not made with Unity. That makes this a bit more difficult, but not all particles one wishes to recreate are from games that have been made with Unity, so at some point one must go off the beaten path.

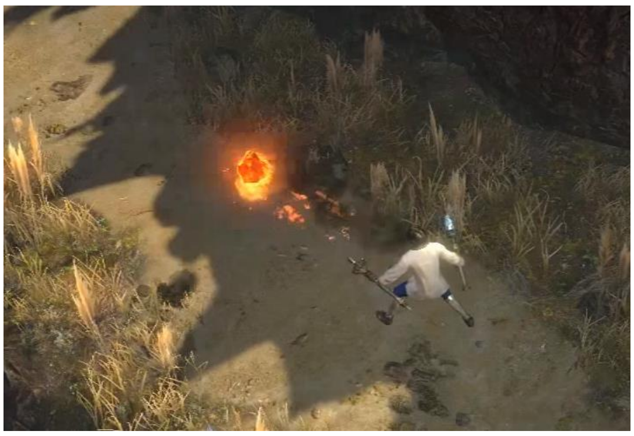


FIGURE 62 Magma Orb projectile (Grinding Gear Games, 2015)

Magma Orb is a straightforward skill, it lobs an orb from the player character in an upwards angle that has a fiery **Trail** on it and it bounces on each hit to the ground until it reaches the maximum number of bounces. The projectile itself could be created with **Mesh** or **Circle** shape or just by using the **Mesh Rendering Mode**.

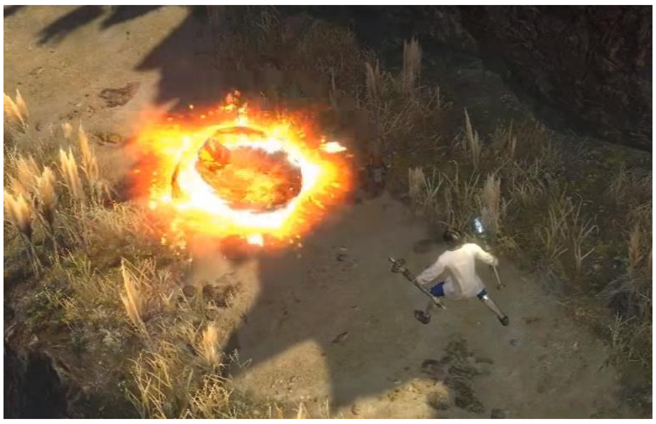


FIGURE 63 Magma Orb ground collision emission (Grinding Gear Games, 2015)

As we can see, with each bounce the orb leaves behind a fire burst, which would most likely make it a system that has the fire bursts set on a **Sub Emitter** to emit on **Collision**. Speaking of collision, this type of effect that bounces would most definitely use the **Collision module** and have appropriate values set on **Bounce** and **Dampen** as while Magma Orb definitely bounces, it also loses speed on each bounce.

5.2 Scoutfly example (Monster Hunter World)

The Scoutfly in Monster Hunter World directs the player to tracks and points of interest alike, working as a guide of sorts. It is a core mechanic in the game, that is visible almost at all times outside of combat. It also changes color depending on the situation, but for the sake of the example we stick to green.

The effect also changes a bit depending on the circumstances, either by emitting more particles or moving faster, therefore we aim to replicate the overall behavior of the system and not the smallest details.

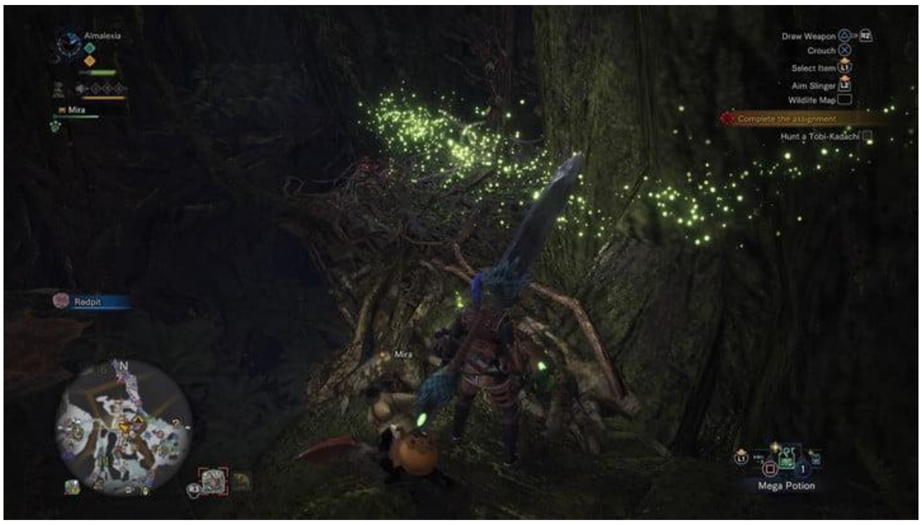


FIGURE 64 MHW Scoutflies in game (Capcom, 2018)

As we can see from figure 64, the effect itself is a trail of particles emitting from a point that moves in the game world and leaves the particles behind. This can be replicated via **Rate over Distance** emission mode and by setting the **Simulation Space** to **World**. The particles are also rather small and do not move that fast, so a low value in both **Start Speed** and **Start Size** is the way to go.

The particles themselves seem to have a bit of randomness in them, this could be replicated by either using **Random Position**, **Noise.** The emission point itself could be either a small **Cone** or a single point.

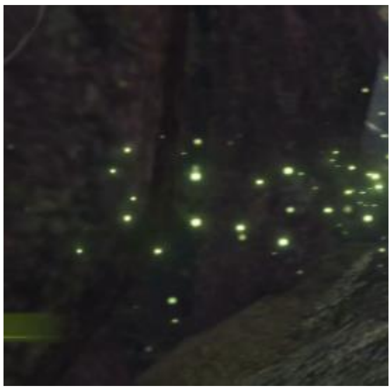


FIGURE 65 Close up of the Scoutfly particles (Capcom, 2018)

From the close up in figure 65, we can see that the particles differ in size, glow slightly and fade out as they die. Now there are multiple ways to approach the size and fading, while the glow can be created in the texture itself. Either the fade is made completely by **Size over Lifetime** or it is bolstered by **Color over Lifetime.** The size differences can also be because of a random size value upon particle birth.

That covers just about the whole effect, now all is that is left to do, is to try out different combinations of options and see what comes closest to the original effect. Below in figure 66, we can see what the effect looks like without **Noise** and **Randomize Direction**. The effect is a bit too uniform compared to the effect in MHW.

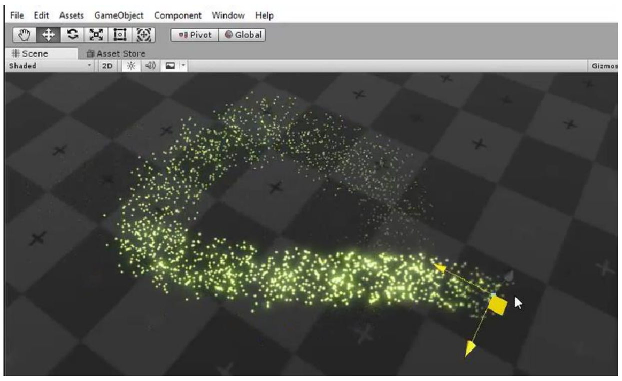


FIGURE 66 Scoutflies made in Unity

After a few tweaks from the previous version, including adding some **Noise** and **Randomize Direction**, we also modified **Start Size** to be a **Random between two Constants** and adjusted a few more values in the over lifetime modules. The emission **Shape** was change to a **Cone** to make the emission less uniform and as we can see, this has brought the behavior much closer to what it is in Monster Hunter World from figure 67.

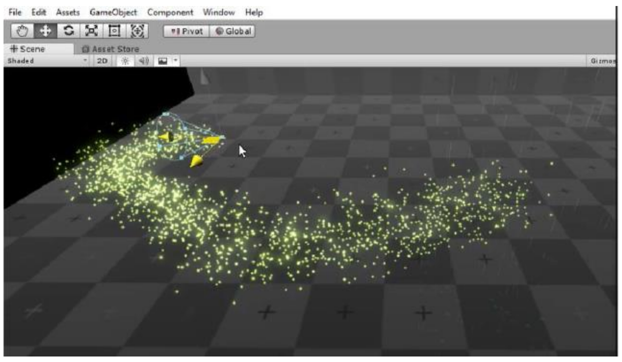


FIGURE 67 Scoutflies made in Unity version 2

The module settings for this particle system and the texture used can be found in appendix 1.

5.3 Pokémon GO Rain

Lastly, we will partly recreate the rain particles in Pokémon GO. From figure 67, we can infer that the raindrops are more than likely capsules or strokes with the tips faded out a bit. Also, we can also see that the raindrops leave a ring expanding outwards as they collide with the ground. So, most likely scenario is that the raindrops have both **Collisions** and **Sub Emitters** enabled. The raindrops are also probably using **Stretched Billboard** rendering mode due to the camera angle.

Also, the raindrops are certainly transparent to some degree and emission area seems to cut off in the top-left corner. I would assume that either a **Rectangle** or **Box** was used for the shape.

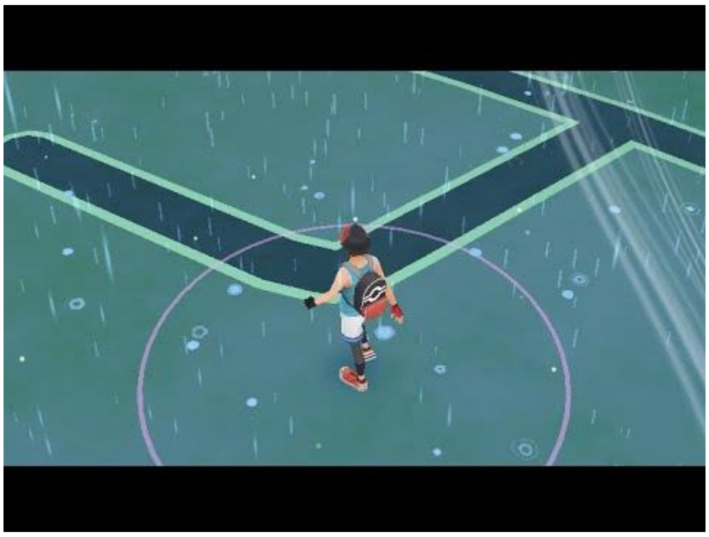


FIGURE 68 Rain in Pokémon GO

The splashes and the rings on the ground are made by a **Sub Emitter** that is either set to **Collision** or **Death** mode. The splashes also seem to be using a **Texture Sheet** to animate the effect. But the rings are a single particle emission that does not move, only grows in size.

In my version, I opted for **Box** shaped emitter that has been turned 180 degrees on the Y-axis and scaled in size to cover a larger area. The raindrops themselves have **Random between two Constants** applied to **Start Size** and **Velocity over Lifetime's** Y-axis to create more randomness in the effect. **3D Start Size** is also used in the raindrops and they also have really low alpha values in their **Color** to more accurately simulate rain that we can see in figure 69.

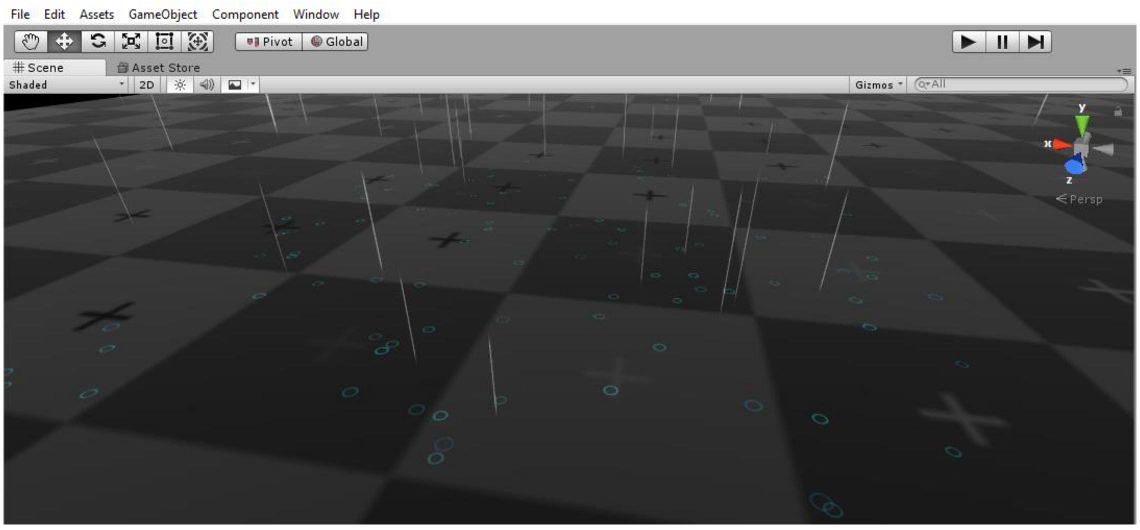


FIGURE 69 Partial Pokémon GO rain in Unity

The **Collision** module in the raindrops has **Kill Speed** set to 100 to kill all particles on ground impact and spawn the **Sub Emitter** upon dying. The sub emitter itself has the **Horizontal Billboard** rendering mode to align it with the ground, a low to high **Size over Lifetime** curve, a **Gradient** with a fade at the end in **Color over Lifetime** and a single particle **Burst** emission. The particle in the **Sub Emitter** has low lifetime and size coupled with no speed.

I did not use a **Texture Sheet** to animate the splash, since I do not have one. That part of the effect has been removed because of it.

The module settings for this particle system can be found in appendix 2. The first four images are for the main particle system and the latter 2 for the sub emitter, the **Renderer** module in the sub emitter is on default settings outside of the material.

As a final reminder, these are not meant to be 100% replicas, just a demonstration of what one could use to replicate these effects, this applies to both the Monster Hunter example and Pokémon GO one.

6 CONCLUSIONS

The Particle System in Unity offers numerous tools to create visual effects in Unity and this thesis only scratched the surface of what you can do with the system. This was no more than a small demonstration of the possibilities of the system by someone who has recently started to learn it. Making visual effects requires many skills and techniques that one hones through years of work, but you have to start from somewhere and hopefully this thesis helps some people find that starting point.

I feel as if the goals for this thesis were met. It will help people at OGL as a basic guide of particles and fill the gap that a lot of people are hesitant to cross when starting out.

The plan for making this thesis was to learn on my own for a month and then start writing based on experience. The plan worked fairly well, but as time went on, it became clear that content needed to be cut by a lot. Time was the true enemy however, since I had no previous experience in the Particle System, everything took that much longer to learn, and I also wasted time making textures for tutorials, when I could have just learned the techniques in the short time I had. Also, reading articles from VFX pros really helped me in thinking about them and breaking them down.

Personally, I feel as if I have achieved my goals of furthering my skills in making visual effects. I started this thesis as a complete novice in making visual effect and over the course of making it, I’ve learned so much. I also learned how much I don’t know, and I look forward to learning about them, however complex and mind boggling they may be.

SOURCES

Unity Manual 2018.1 2018. Particle Systems.
Available: https://docs.unity3d.com/Manual/ParticleSystems.html
Accessed at 25.9.2018

Travis 2017. Beginning VFX Artist – Advice for beginners, from a beginner. Available: https://realtimevfx.com/t/beginning-vfx-artist-advice-for-beginners-from-a-beginner/3081
Accessed at 27.9.2018.

Kris Decker 2016. The fundamentals of understanding color theory. Available: https://99designs.com/blog/tips/the-7-step-guide-to-understanding-color-theory/
Accessed at 27.9.2018

League of Legends VFX Discipline 2017. The complete guide to creating visual effects within. Available:
https://nexus.leagueoflegends.com/wpcontent/uploads/2017/10/VFX_Styleguide_final_public_hid
pjqwx7lqyx0pjj3ss.pdf
Accessed at 27.9.2018

Anthony Uccello 2018. Introduction to Unity: Particle Systems.
Available: https://www.raywenderlich.com/138-introduction-to-unity-particle-systems
Accessed at 05.10.2018

David Finseth 2018. Creating 2D Particle Effects in Unity3D.
Available: https://blog.kongregate.com/creating-2d-particle-effects-in-unity3d/
Accessed at 07.10.2018

Matt Schwarz 2018. From Realism to Stylization: Game VFX Production. Available:
https://80.lv/articles/from-realism-to-stylization-game-vfx-production/
Accessed at 11.10.2018

Unity Manual 2018.1 2018. Materials, Shaders & Textures.
Available: https://docs.unity3d.com/Manual/Shaders.html
Accessed at 15.10.2018

Brackeys 2018. Everything to know about the PARTICLE SYSTEM.
Available: https://www.youtube.com/watch?v=FEA1wTMJAR0
Accessed at 21.10.2018

Gabriel Aguiar Prod 2018. Unity 2018 – Game VFX – 10 Performance Improvements Tips
Available: https://www.youtube.com/watch?v=JqWvJK1uFn8
Accessed at 21.10.2018

Gabriel Aguiar Prod 2018. Unity 2018 – Game VFX – UI / User Interface Effects
Available: https://www.youtube.com/watch?v=hiRdux33UCs
Accessed at 23.10.2018

Michał Piątek 2018. Unity3D Particle Shaders – The simplest shader. Available:
http://michalpiatek.com/2017/06/08/unity3d-particle-shaders-the-simplest-shader/
Accessed at 28.10.2018

Unity Manual 2018.1 2018. Art Asset best practice guide. Available:
https://docs.unity3d.com/Manual/HOWTO-ArtAssetBestPracticeGuide.html
Accessed at 28.10.2018

Jordan Stevens 2016. Particle Shading.
Available: http://www.jordanstevenstechart.com/particle-shading
Accessed at 29.10.2018.

Alan Zucconi 2015. A Gentle Introduction to Shaders. Available:
https://unity3d.com/learn/tutorials/topics/graphics/gentle-introduction-shaders
Accessed at 16.11.2018

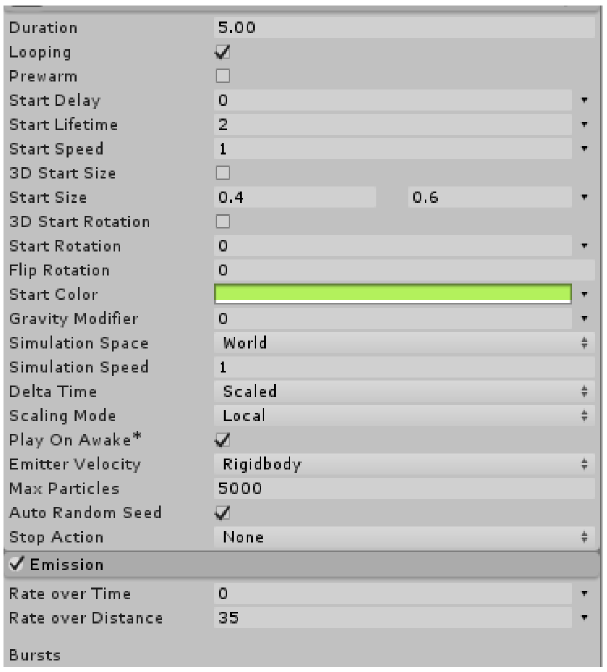
Matt Shell/Unity 2017. Controlling Particles Via Script. Available:
https://www.youtube.com/watch?v=JRa2g3vgzBo

Accessed at 20.11.2018

Matt Shell/Unity 2017. Visual Effects with Particle Systems. Available:
https://www.youtube.com/watch?v=JRa2g3vgzBo
Accessed at 21.11.2018

Karl Jones 2016. Particle System Modules – FAQ. Available:
https://docs.unity3d.com/uploads/ExpertGuides/Particle_System_Modules.pdf
Accessed at 21.11.2018

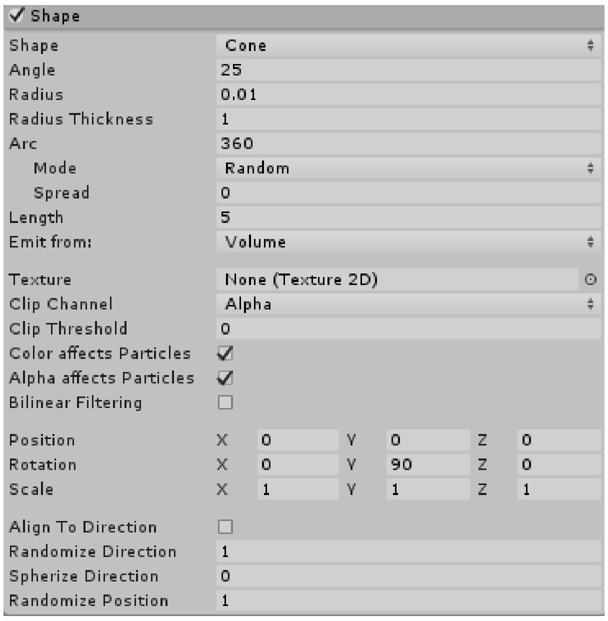
Francisco Garcia-Obledo Ordóñez 2017. VFX for Games Explained. Available:
https://80.lv/articles/vfx-for-games-explained/
Accessed at 26.11.2018



Duration	5.00	
Looping	✓	
Prewarm	☐	
Start Delay	0	
Start Lifetime	2	
Start Speed	1	
3D Start Size	☐	
Start Size	0.4	0.6
3D Start Rotation	☐	
Start Rotation	0	
Flip Rotation	0	
Start Color	🟩	
Gravity Modifier	0	
Simulation Space	World	
Simulation Speed	1	
Delta Time	Scaled	
Scaling Mode	Local	
Play On Awake*	✓	
Emitter Velocity	Rigidbody	
Max Particles	5000	
Auto Random Seed	✓	
Stop Action	None	

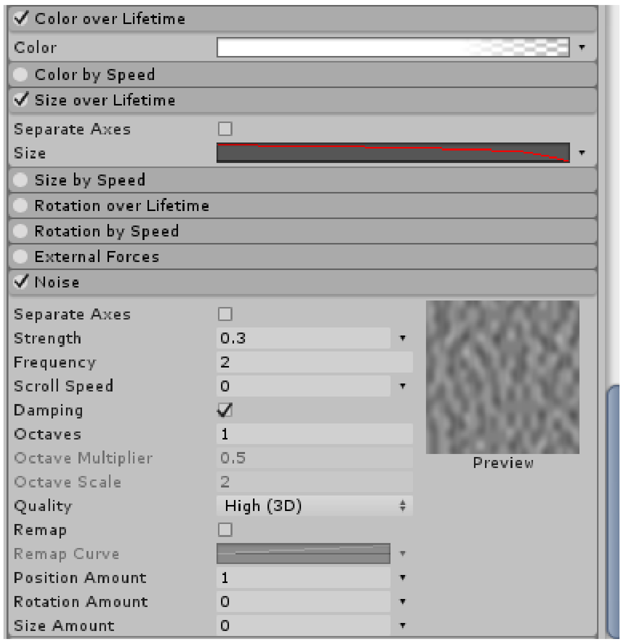
✓ Emission

Rate over Time	0
Rate over Distance	35
Bursts	

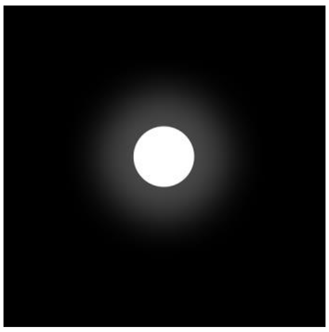


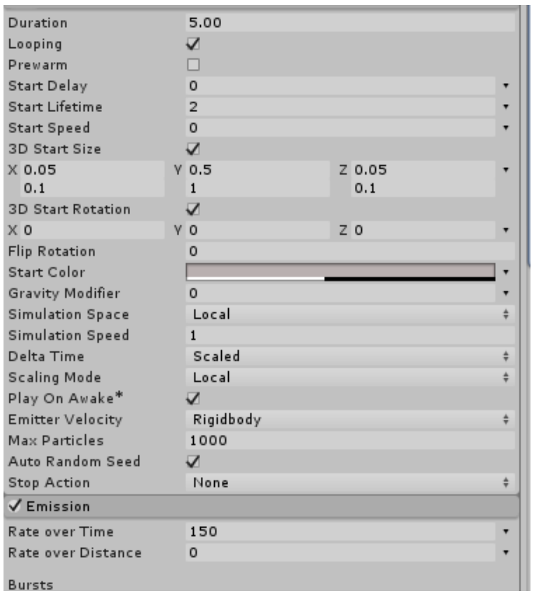
✓ Shape

Shape	Cone					
Angle	25					
Radius	0.01					
Radius Thickness	1					
Arc	360					
Mode	Random					
Spread	0					
Length	5					
Emit from:	Volume					
Texture	None (Texture 2D)					
Clip Channel	Alpha					
Clip Threshold	0					
Color affects Particles	✓					
Alpha affects Particles	✓					
Bilinear Filtering	☐					
Position	X	0	Y	0	Z	0
Rotation	X	0	Y	90	Z	0
Scale	X	1	Y	1	Z	1
Align To Direction	☐					
Randomize Direction	1					
Spherize Direction	0					
Randomize Position	1					



✓ Color over Lifetime	
Color	
Color by Speed	
✓ Size over Lifetime	
Separate Axes	☐
Size	
Size by Speed	
Rotation over Lifetime	
Rotation by Speed	
External Forces	
✓ Noise	
Separate Axes	☐
Strength	0.3
Frequency	2
Scroll Speed	0
Damping	✓
Octaves	1
Octave Multiplier	0.5
Octave Scale	2
Quality	High (3D)
Remap	☐
Remap Curve	
Position Amount	1
Rotation Amount	0
Size Amount	0



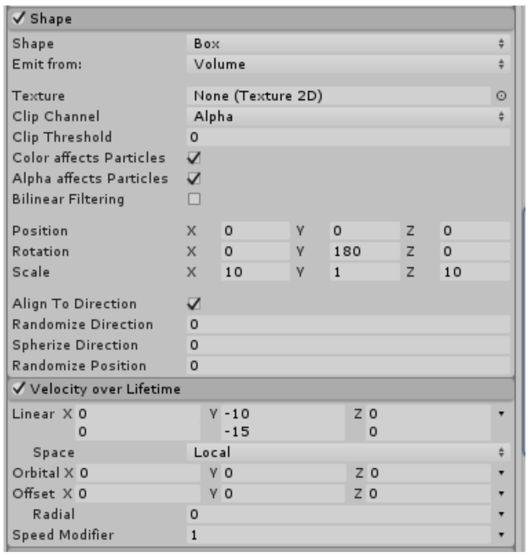


Duration	5.00		
Looping	✓		
Prewarm	☐		
Start Delay	0		
Start Lifetime	2		
Start Speed	0		
3D Start Size	✓		
	X 0.05 / 0.1	Y 0.5 / 1	Z 0.05 / 0.1
3D Start Rotation	✓		
	X 0	Y 0	Z 0
Flip Rotation	0		
Start Color			
Gravity Modifier	0		
Simulation Space	Local		
Simulation Speed	1		
Delta Time	Scaled		
Scaling Mode	Local		
Play On Awake*	✓		
Emitter Velocity	Rigidbody		
Max Particles	1000		
Auto Random Seed	✓		
Stop Action	None		

✓ Emission

Rate over Time	150
Rate over Distance	0

Bursts

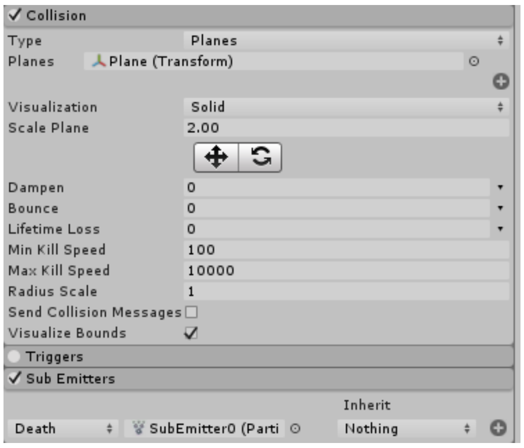


✓ Shape

Shape	Box		
Emit from:	Volume		
Texture	None (Texture 2D)		
Clip Channel	Alpha		
Clip Threshold	0		
Color affects Particles	✓		
Alpha affects Particles	✓		
Bilinear Filtering	☐		
Position	X 0	Y 0	Z 0
Rotation	X 0	Y 180	Z 0
Scale	X 10	Y 1	Z 10
Align To Direction	✓		
Randomize Direction	0		
Spherize Direction	0		
Randomize Position	0		

✓ Velocity over Lifetime

Linear	X 0 / 0	Y -10 / -15	Z 0 / 0
Space	Local		
Orbital	X 0	Y 0	Z 0
Offset	X 0	Y 0	Z 0
Radial	0		
Speed Modifier	1		



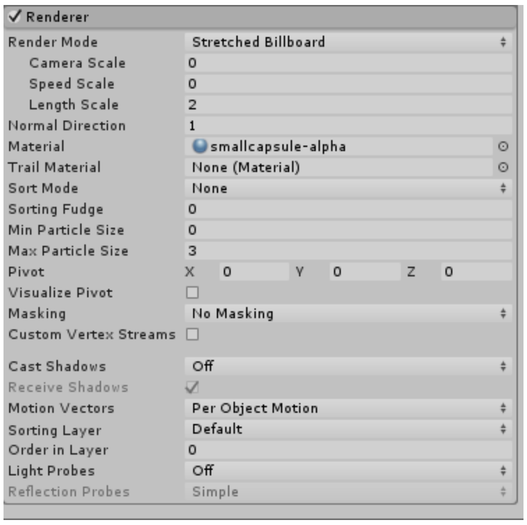
✓ Collision

Type	Planes
Planes	Plane (Transform)
Visualization	Solid
Scale Plane	2.00
Dampen	0
Bounce	0
Lifetime Loss	0
Min Kill Speed	100
Max Kill Speed	10000
Radius Scale	1
Send Collision Messages	☐
Visualize Bounds	✓

Triggers

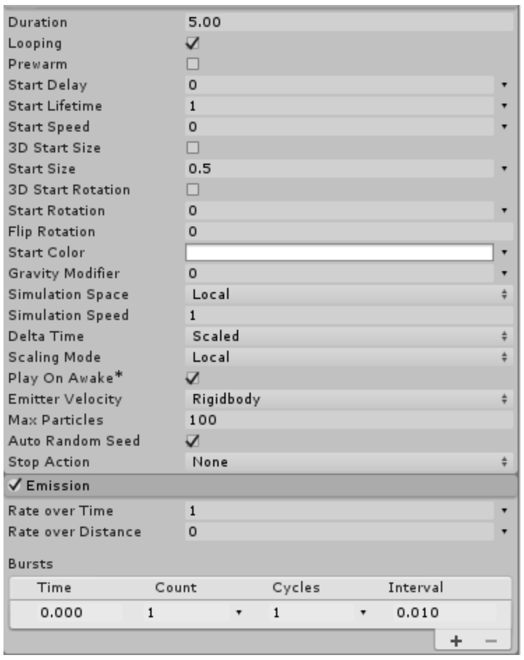
✓ Sub Emitters

		Inherit
Death	SubEmitter0 (Parti	Nothing



✓ Renderer

Render Mode	Stretched Billboard
Camera Scale	0
Speed Scale	0
Length Scale	2
Normal Direction	1
Material	smallcapsule-alpha
Trail Material	None (Material)
Sort Mode	None
Sorting Fudge	0
Min Particle Size	0
Max Particle Size	3
Pivot	X 0 Y 0 Z 0
Visualize Pivot	☐
Masking	No Masking
Custom Vertex Streams	☐
Cast Shadows	Off
Receive Shadows	✓
Motion Vectors	Per Object Motion
Sorting Layer	Default
Order in Layer	0
Light Probes	Off
Reflection Probes	Simple



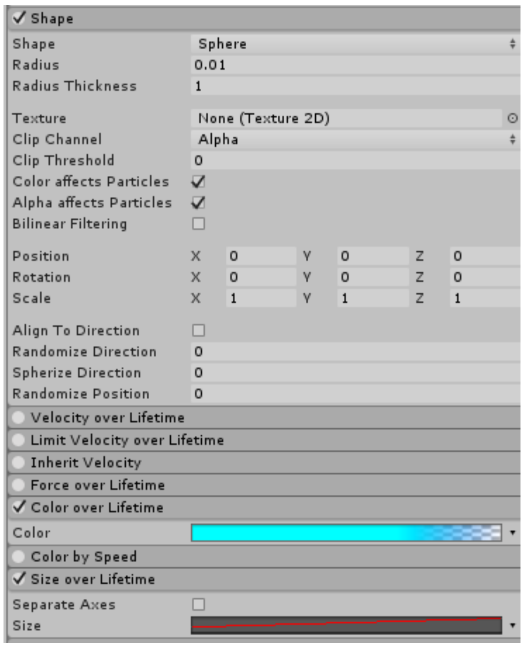
Duration	5.00
Looping	✓
Prewarm	☐
Start Delay	0
Start Lifetime	1
Start Speed	0
3D Start Size	☐
Start Size	0.5
3D Start Rotation	☐
Start Rotation	0
Flip Rotation	0
Start Color	⬜
Gravity Modifier	0
Simulation Space	Local
Simulation Speed	1
Delta Time	Scaled
Scaling Mode	Local
Play On Awake*	✓
Emitter Velocity	Rigidbody
Max Particles	100
Auto Random Seed	✓
Stop Action	None

✓ Emission

Rate over Time	1
Rate over Distance	0

Bursts

Time	Count	Cycles	Interval
0.000	1	1	0.010



✓ Shape

Shape	Sphere
Radius	0.01
Radius Thickness	1
Texture	None (Texture 2D)
Clip Channel	Alpha
Clip Threshold	0
Color affects Particles	✓
Alpha affects Particles	✓
Bilinear Filtering	☐
Position	X 0 Y 0 Z 0
Rotation	X 0 Y 0 Z 0
Scale	X 1 Y 1 Z 1
Align To Direction	☐
Randomize Direction	0
Spherize Direction	0
Randomize Position	0

Velocity over Lifetime

Limit Velocity over Lifetime

Inherit Velocity

Force over Lifetime

✓ Color over Lifetime

Color	🟦

Color by Speed

✓ Size over Lifetime

Separate Axes	☐
Size	