Mercury Particle Engine Manual

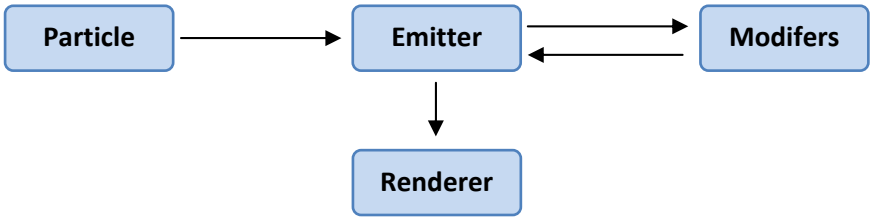
Table of Contents

Overview

Mercury Particle Engine (MPE) is used to easily add visual effects to games based on the XNA platform. Mercury Particle Engine can rapidly create particle effects in your game by taking care of the rendering and giving you the ability to plug in modifiers to change the behavior of each single particle.

Engine structure

The structure of the engine is quite simple. It consists of particles, emitters, modifiers and a renderer.



As you can see, the emitter is the main worker in this case. Particles are created to be used in the emitter, the emitter then send each particle to the modifiers. The modifiers send back the particles which then are send to the renderer and drawn on the screen.

The structure is quite flexible and it's easy to extend with new features.

Particle

Without this, there is no particle engine. It's the most important part of the engine, but also one of the simplest. A particle does not know about the world around it or other particles. It knows about its own position, rotation, scale and color. They are all properties that make sure we can draw the particles on the screen. It does also have some knowledge about velocity, age and momentum; they are all used to simulate some sort of movement and to make sure that the particles gets removed from the screen after some time.

Example: Imagine an explosion. An explosion consists of particles (or can be simulated by particles) that are in different colors and size, but also scattered around and rotated randomly. The particles in an explosion also have a velocity going away from the center of the explosion, but also an age that defines how long the explosion lasts. So how do we make an explosion in Mercury Particle Engine? Well, first we need to know about emitters.

Emitters

Emitters are simply a way of releasing particles. They have several different shapes that the released particles can form. Mercury Particle Engine already has some predefined shapes out of the box, but you are free to create your own by using the base class Emitter.

Emitters have several important properties that define the characteristics of the emitter. You can see a description of them right here:

Budget
This is one of the most important properties. You need to set the budget on all emitters to tell them how many active particles they can have. If you set this to 1000, the emitter can't have more than 1000 active particles at one time.

ReleaseQuantity
This defines the quantity of particles released. Whenever you update the particle engine some particles will be released; this amount is defined by the value you set ReleaseQuantity to. With a budget of 3000 and a ReleaseQuantity of 10, you will get a light steady stream of particles; you can control the density of the particles by having a higher ReleaseQuantity.

Term
This is used to tell the emitter how long the released particles will stay active. Most particles will fade away after some time and thus we need a mechanism for keeping track of active particles. Term combined together with age (on each particle) is essentially that mechanism.

ParticleTexture
Particles need some sort of shape and texture. You could just do with a single pixel, but to get more complex and real particle simulation, you will need a shape and texture. This property defines the texture applied to the particles.

Modifiers
This is also one of the more important ones. We will describe modifiers later in its own chapter. This property is a collection of the working modifiers on the emitter.

Others
We also have **ReleaseSpeed**, **ReleaseColour**, **ReleaseOpacity**, **ReleaseScale** and **ReleaseRotation**. They all control the initial properties of the particle when it's released.

Emitter usage

To make emitters work, it is important to make sure it's properly initialized and loaded. Emitters will not work unless they have been initialized and loaded first.

Emitters have an **Initialize()** method that makes sure to do all the initialization of the emitter, such as creating the particle arrays. If you use XNA, you can just initialize the emitter in the XNA built-in method conveniently called: Initialize(). Emitters also have a method called **LoadContent()** that is responsible of loading the emitter texture. Again, if you run XNA, just call this method inside the XNA method: LoadContent()

To make an emitter release particles, you have the **Trigger()** method on the emitter object. Trigger() takes in a position of where the release should happen. This release position is then enqueued for later when we update the emitter. It is made that way to make it possible to release particles several places on the screen simultaneously.

As mentioned before, emitters need to be updated. The update makes sure that the particles are updated with new positions and released at the correct positions. This method is called **Update()** and should be called inside your game loop (Called Update() in XNA)

Different kinds of emitters

Mercury Particle Engine comes with several different emitters out of the box. The default emitter (simply called Emitter) releases the particles in a single point. Sometimes we want more than a single point, we want to release the particles in different shapes, such as circles and rectangles. Here is a list of out-of-the-box emitters:

LineEmitter
Distributes all particles randomly on a single line. You can control 4 properties on the line emitter: Length, Angle, Rectillinear and Frame. Length and Angle should be self-explanatory.

The Rectillinear property controls if the emitted particles should decent in a linear manner. The Frame property is used to control if the emitter should emit particles both ways on the Y-axis.

This emitter is great to create something like snow. You set the length of the emitter, to be the length of the screen and that it should be horizontal. All particles (snowflakes) are then released in a line on top of the screen. We would need something like gravity too, to make the particles decent. More on that later in the modifiers chapter.

CircleEmitter
This emitter releases particles randomly distributed in a circle. It has two properties you can change to control the behavior of the emitter: Ring and Radiate.

Ring tells the circle emitter to only create particles on the edge of the circle. You can set this to false to create a filled circle.

Radiate makes sure that the particles radiate away from the center of the circle.
The circle emitter is good for explosions and the like. Depending on the kind of explosion, you just enable or disable Ring and Radiate. The different release patterns can be seen here:

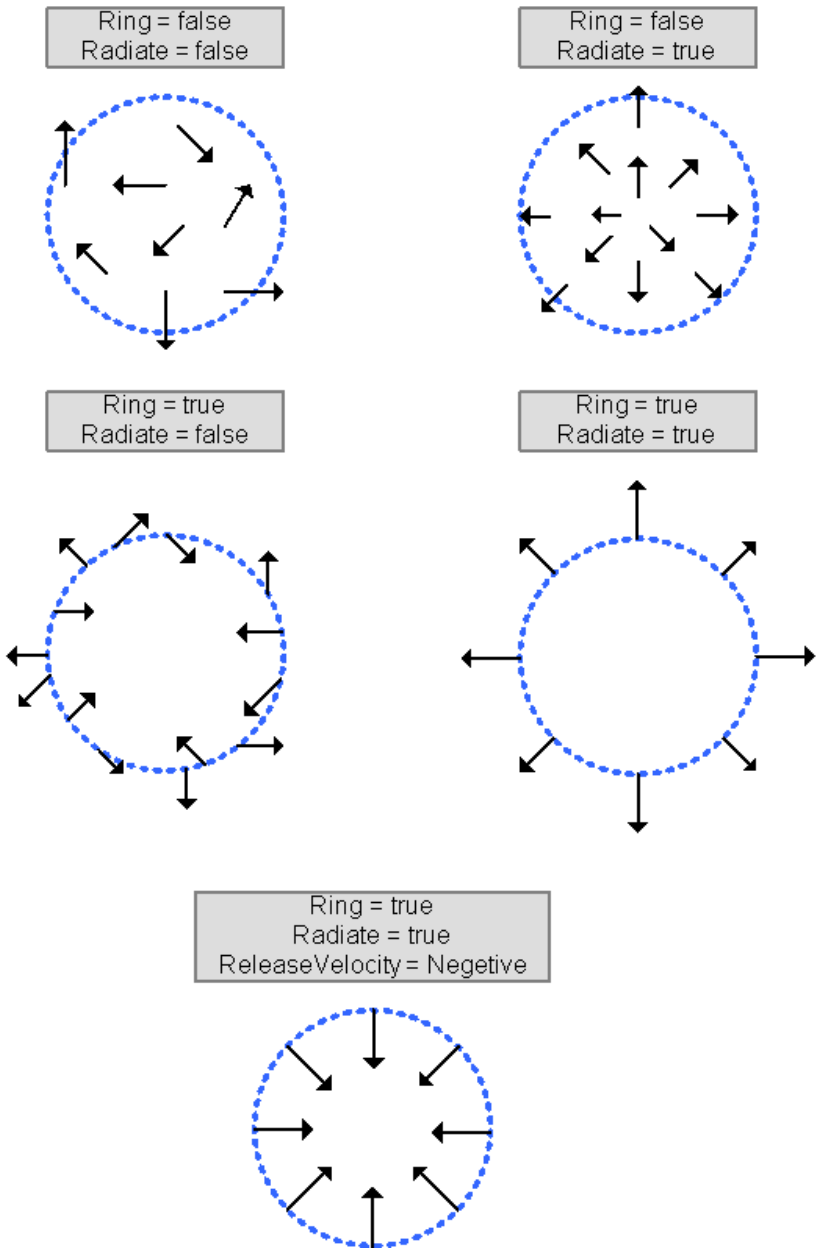


Figure 1 - CircleEmitter release patterns with different properties

ConeEmitter

The ConeEmitter sprays the particles out in the specified direction with the specified angle. The emitted particles will take the shape of a spray/cone.

RectEmitter

Rectangle emitter releases the particles in the shape of a rectangle. You control the width and height of the rectangle, but just as the CircleEmitter, you also have the possibility of making the particles spawn only on the edge of the rectangle. You enable this behavior by enabling Frame on the RectEmitter.
Imagine a checkbox inside your game. You could make a nice glow or pulsating effect around this checkbox by using the RectEmitter. To make it pulsate, you will also have to use modifiers, more on this later.

Modifiers

Modifiers are small pieces of logic that alter the particles before they are send further down the system to be drawn. Modifiers can change any of the particle properties on the fly, this includes position, rotation, color, scale, velocity and others. Mercury Particle Engine has several modifiers out of the box that easily can be activated to create a dynamic behavior. You can see a short description of them here:

ColorModifier

This modifier changes the color from an initial value to an ultimate value. It does this over the lifetime of the particle. It's great for making particles in an explosion fade from bright yellow to dark orange.

RandomColourModifier

This modifier changes the color of the particle to a random color.

OpacityModifer

The Opacity modifier changes the opacity of the particles from an initial value to an ultimate value. Just as the Color modifier, it does this over the lifetime of the particle. Again, if we take the explosion example, the particles can fade away as the explosion particles cool down.

LinearGravityModifier

This modifier enables gravity with the defined gravity vector (strength and direction). With this enabled particles automatically fall to the ground, or any other direction you specify in the vector.

RadialGravityModifer

Compared to the linear gravity modifier, this gravity modifier creates a gravity field at the specified position. You can define the strength and radius of the modifier by using the Strength and Radius properties of the modifier.

RotationModifier

Changes the rotation with a specified rotation rate.

Modifier usage

To use a modifier, you first have to create it and then add it to the emitter. The emitter is responsible of sending its particles to the modifiers. You simply create a LinearGravityModifer like this:

```
LinearGravityModifier linearGravityModifier = new LinearGravityModifier();
linearGravityModifier.Gravity = new Vector2(0, 128);
Emitter.Modifiers.Add(linearGravityModifier);
```

That is it. The rest is done automatically inside the engine.

Renders

Mercury Particle Engine supports several renders for drawing the particles. It's a pretty flexible structure where you easily can create your own render for a different platform than Mercury Particle Engine supports. We currently only support rendering on the XNA platform. The rendering structure is split into 2 renders:

PointSpriteRenderer
This renderer uses point sprites to draw the particles. Point sprites are feature that came with DirectX and offers a more effective way of drawing the particles. Before we had to send 4 vertices to the graphics card, but now we only have to send one single vertex. This improves performance a lot but is also only supported on newer graphics (DirectX 8 enabled graphics cards).

The PointSpriteRenderer uses the graphics device directly by calling DrawUserPrimitives() in on the GraphicsDevice object. It's very efficient and can handle a lot of particles.

SpriteBatchRenderer
This renderer uses the XNA SpriteBatch to draw the particles. Using the SpriteBatch brings all the features that the SpriteBatch normally have; such as batching of textures and sorting of textures. It's not really necessary in most cases, but we provide you with the option to use it.

Renderer usage
Renderers need to be loaded and then on each update, it needs to be drawn. To load a renderer, you just call **LoadContent()** on the renderer object. Depending on the type of renderer, this method is used to load content specific to the renderer.

On each update, you need to draw the emitters to the screen by using the renderer. This is simply done by calling **Renderer.RenderEmitter()** and give it an emitter as a parameter.

Performance
Mercury Particle Engine has been developed with performance in mind. We really try to squeeze every bit of performance out of the code by create an effective and optimized structure that is easily extendable. It can update and draw about 80,000 active particles on a dual core Athlon processor 3GHz. Even though it's a very high number, we hope to increase the active particle count in the future.

The active particle count depends on your computer and what kind modifiers you have enabled. Just drawing the particles without any modifiers will yield the best performance. As soon as you add some modifiers, the active particle count will decrease from the added load to the engine.

So how do we gain better performance? Well, modifiers are of course needed to create dynamic behavior and an explosion effect looks the best when a lot of particles are used. To increase performance, there is simply nothing to do but to decrease the budget on your emitters and use fewer modifiers.

Multithreading
The nature of the particle engine makes it possible to use multithreading in an efficient manner. You have several possibilities of making MPE multithreaded. You could simply create multiple particle engine instances and put each of them on their own thread. Another way is to create a thread per emitter. Have in mind that multithreading have some overhead; creating 2 threads instead of one will not give you double as many active particles to work with. Experiment a little with it.

Offloading to GPU

GPUs work differently than CPUs. GPUs are great for parallel processing and have multi-core design that greatly improves their parallel processing efficiency. Nvidia have a SDK called CUDA (Compute Unified Device Architecture) that you can use to access the functionality of the GPU (Nvidia cards only).

It is possible to combine MPE and Nvidia's CUDA, but Mercury Particles Engine does not support processing on the GPU out of the box.

Examples

Let's create some stunning effects! In this chapter you will learn how to create 2 different effects: Explosions and snow. They are common effects in most games and an essential part of you want to create fun and dynamic games. Let us get started.

Explosions

To create an explosion, we must think about what an explosion consist of. Imagine a fireworks explosion, it has a relatively high particle count, high velocity, different colors, and the particles fade after some time. If we put that into a list it would look like this:

- Large particle count
- High velocity
- Variable color (Red to orange)
- Variable opacity (Particles burn out)

Now we need to find the correct emitters and modifiers to get the desired effect.

Emitter

Explosions are circular in size and they start from a single point (the firework rocket) and spread outwards. By looking at **Figure 1** from the Emitter chapter, we can determine that we need a CircleEmitter with Ring = false and Radiate = true. Code looks like this:

```
emitter = new CircleEmitter();
emitter.Budget = 10000;
emitter.Radiate = true;
emitter.Radius = 10f;
emitter.ReleaseQuantity = 1000;
emitter.ReleaseScale = 10f;
emitter.ReleaseSpeed = new VariableFloat { Anchor = 50f, Variation = 5f };
emitter.ReleaseColour = Color.Yellow.ToVector3();
emitter.ReleaseOpacity = 1f;
emitter.Ring = false;
emitter.Term = 3f;
```

Now we have solved the following requirements:

- Large particle count - √
- High velocity - √
- Variable color (Red to orange)

- Variable opacity (Particles burn out)

Modifiers

Now we need to add some dynamics to our explosion. We still need variable color and variable opacity. This is easily done with modifiers. We first start with a ColorModifier:

```
ColorModifier colorModifier = new ColorModifier();
colorModifier.InitialColour = Color.Yellow.ToVector3();
colorModifier.UltimateColour = Color.OrangeRed.ToVector3();

emitter.Modifiers.Add(colorModifier);
```

Then we take a look at variable opacity:

```
OpacityModifier opacityModifier = new OpacityModifier();
opacityModifier.Initial = 1.0f;
opacityModifier.Ultimate = 0f;

emitter.Modifiers.Add(opacityModifier);
```

So how far are we with our particle effect?

- Large particle count - √
- High velocity - √
- Variable color - √
- Variable opacity - √

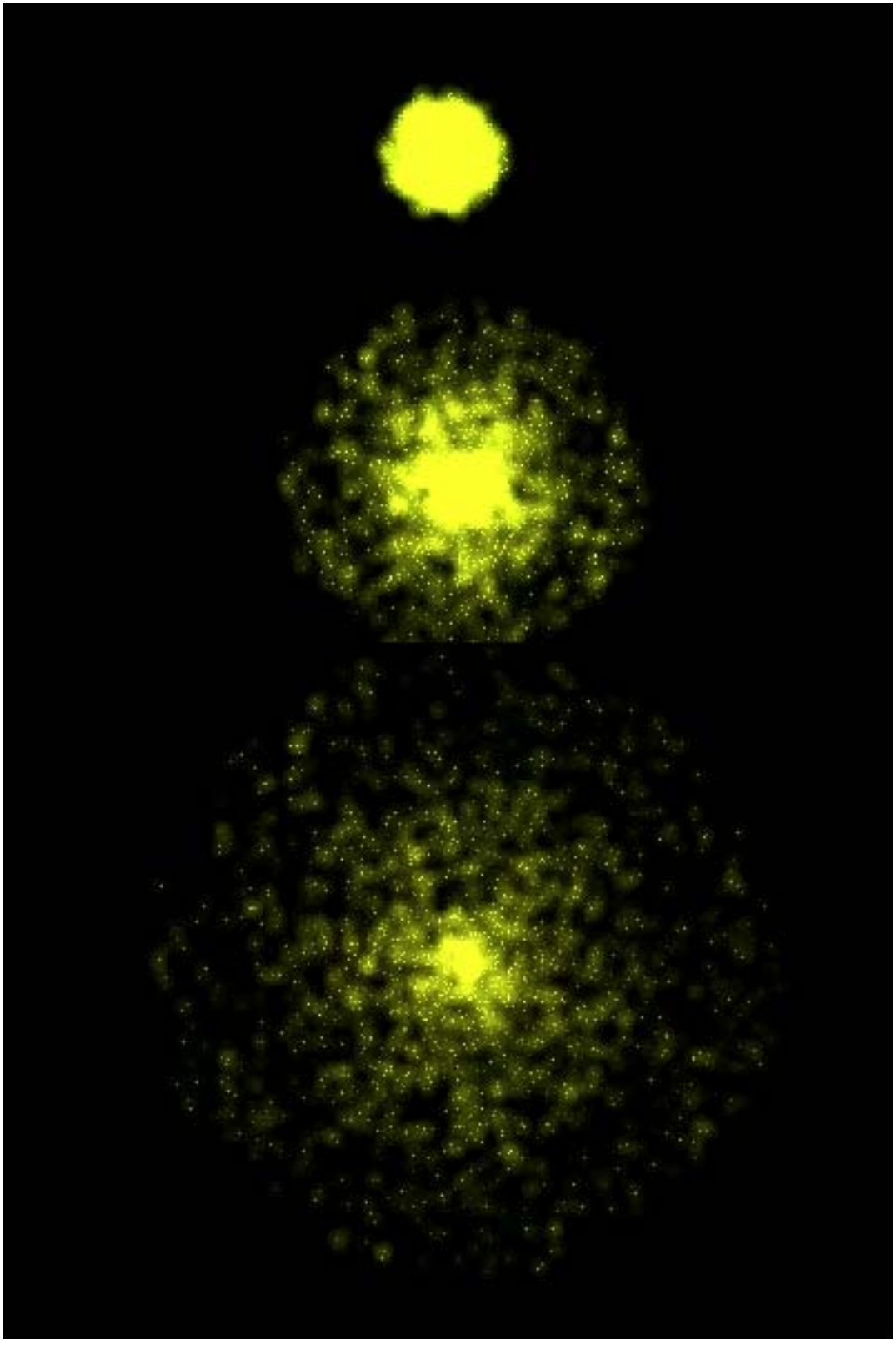
We are done! Just one thing left to do: Draw it to the screen.

Rendering

We also need to add a renderer. This is used to draw the particles to the screen. We simply use the PointSpriteRenderer like this:

```
renderer = new PointSpriteRenderer();
renderer.GraphicsDeviceService = graphics;
renderer.BlendMode = SpriteBlendMode.Additive;
```

In the end you should end op with something like this (Colors don't match because of capturing software):



Snow

Just like the explosion effect, we need to think about what kind of behavior snow has. First of all it is white, scattered around and it falls to the ground slowly. If we setup this in a list, it would look like this:

- White color
- Scattered around
- Falls to the ground slowly

Emitter

We would like the whole game screen to be snowing. To do this we create a LineEmitter that is the same size (width) as the screen and has no angle:

```
emitter = new LineEmitter();
emitter.Budget = 5000;
emitter.Length = 800;
emitter.Angle = 0f;
emitter.ReleaseQuantity = 1;
emitter.ReleaseScale = 30f;
emitter.ReleaseSpeed = new VariableFloat { Anchor = 1f, Variation = 10f };
emitter.ReleaseColour = Color.White.ToVector3();
emitter.ReleaseOpacity = 1f;
emitter.Term = 25f;
```

By now we have already satisfied some of our requirements:

- White color - √
- Scattered around - √
- Falls to the ground slowly

Modifiers

Now we need the snow to fall to the ground. We use the gravity modifier for this:

```
LinearGravityModifier gravity = new LinearGravityModifier();
gravity.Gravity = new Vector2(0, 5);

emitter.Modifiers.Add(gravity);
```

That's basically it. We have not created snow with all our requirements:

- White color - √
- Scattered around - √
- Falls to the ground slowly - √

All we need to do now is create a renderer that and draw the particles to the screen.

Renderer

```
renderer = new PointSpriteRenderer();
renderer.GraphicsDeviceService = graphics;
renderer.BlendMode = SpriteBlendMode.Additive;
```

In the end you should end up with something like this: