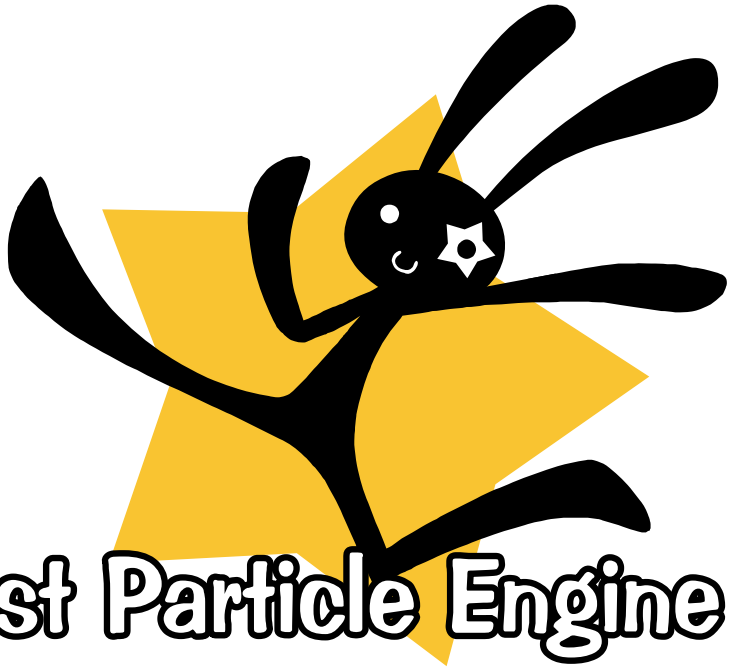


Stardust Particle Engine Manual

Version 1.0

By CJ Cat (Allen Chou)

http://code.google.com/p/stardust-particle-engine

Last Updated: Nov 5, 2009

Table of Contents

License

Stardust Particle Engine is developed by Allen Chou and released under the MIT license, as stated below:

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

About The Author

Name	Allen Chou
Internet ID	CJ Cat (or just CJ)
Birthday	Jul 19, 1988
Blog	http://cjcat.blogspot.com
E-mail	cjcat2266@gmail.com
Occupation	Undergraduate student of The Electrical Engineering Department of National Taiwan University

↑ Yeah, that's me having my delicious lunch on my lovely campus :P

Although I've been playing with Flash since version 3 when I was in elementary school, my actual ActionScript programming career started when I entered college three years ago. I started self-learning ActionScript 2.0 through books in the first semester, and 3.0 the second semester. Since then, I've been working on several ActionScript 3.0 projects, and having part-time freelance jobs with some RIA agencies.

I'm interested in particle effects in particular, and have done quite a lot of research on the subject. Actually, I have another particle engine project called Emitter (this happens to be a bad name for a project, since the term **emitter** is widely used in many realms).

My biggest project so far is Stardust Particle Engine, and this is its manual. I hope I can help you understand Stardust in a very easy and comfortable way, instead of going through a very serious and dull text.

I appreciate you for trying Stardust. This is very encouraging. I'll do my best to maintain this project and update as frequently as I can.

You can support this project by donating.
Click this button to donate: Donate

Allen Chou
Aug 17, 2009

About Stardust Particle Engine

Google Code Project Homepage:
http://code.google.com/p/stardust-particle-engine

What Is A Particle Engine?

Particle engines are also referred to as particle systems. Let's take a look at the ***particle system*** definition on Wikipedia:

> *The term **particle system** refers to a computer graphics technique to simulate certain fuzzy phenomena, which are otherwise very hard to reproduce with conventional rendering techniques. Examples of such phenomena which are commonly replicated using particle systems include fire, explosions, smoke, moving water, sparks, falling leaves, clouds, fog, snow, dust, meteor tails, hair, fur, grass, or abstract visual effects like glowing trails, magic spells, etc.*
> *While in most cases particle systems are implemented in three dimensional graphics systems, two dimensional particle systems may also be used under some circumstances.*

Whew. That's a long statement, but you've probably noticed some keywords: fire, explosions, smoke, moving water, sparks, etc. These effects can be seen in most movies and video games. Actually these effects have a general name, which is not hard to guess if you're readying this manual: particle effects.

I have my own definition for what qualify as particle effects:

> *Particle effects are visual effects consisted of visual entities which usually come in large quantity, sharing similar but not exactly same appearance and behaviors.*

Particle engines are tools for creating particle effects, and Stardust comes in the flavor of an ActionScript 3.0 API.

Why Using A Particle Engine?

The most common particle effects are fire, water, and smoke, which as mentioned before can be seen in most movies and video games.

If you're developing a game, it's very likely that your game needs particle effects to add flavor to the overall visual display. This is when you need a particle engine.

For some experienced programmers, creating particle effects from scratch is probably not difficult, but it usually involves massive coding and a lot of debugging. Using a particle engine can greatly reduce your time spent on this time-consuming routine. A particle engine allows you to be concerned with only the high-level design of the particle effects, such as setting a particle emitter's position, tweaking a gravity field's parameters, or placing obstacles that block particles. You don't need to be worried about the underlying particle logic, since it's taken very good care of by the particle engine.

Why I Created Stardust

I'm not a pro ActionScript developer. Although I've been writing ActionScript codes for years, I still can't bear looking at a screen full of pure-text source codes, because it just gives me a serious headache.

I have been in many scenarios where I've needed to create particle effects for my projects, and the thought of building the effects from scratch alone is killing me. That's why I decided to create a particle engine, to wrap up all the messy codes in to an abstract package and end this coding nightmare for good. As a result, Stardust was born.

Feature Overview

Stardust Supports 2D And 3D

Stardust supports both 2D and 3D particle effects.

Stardust Is Easy to Extend

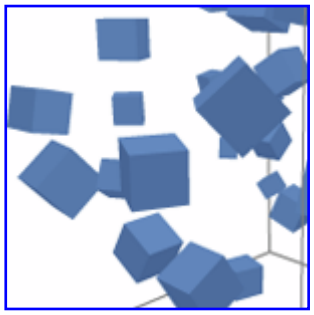
Stardust is designed to be extended easily. The abstract class structure was inspired by Flint Particle System, another ActionScript 3.0 particle engine. By extending the `Action` class, you can easily create custom particle behaviors. By extending the `Renderer` class, you can make Stardust collaborate with other 3D engines.

Stardust Includes Several 3D Engine Extensions

Stardust already includes extensions for several 3D engines, including ZedBox, Papervision3D, and ND3D. You can extend Stardust for whatever engines you like if you're not using the aforementioned engines.

Here's an example of the ZedBox extension.

And here's an example of Stardust combined with Papervision3D.



Stardust Includes A Native 3D Engine

There's another option for creating 3D particle effects, without using other 3D engines: Stardust has its own built-in 3D engine (Stardust 3D). Actually, it's a light-weighted version of ZedBox, my own 3D billboard engine. The only difference between these two engines in terms of functionality is that Stardust 3D does not support hierarchical object structure.

3D engines like Paperivsion3D and ND3D usually draw a display object to a bitmap and then apply this bitmap to particles as a material. This would result in performance issue if the display object's size is large, even if it merely contains simple vector graphics.

Stardust 3D and ZedBox do not draw the display object to a bitmap. They directly add the display object to the display list, so the display object is still rendered by Flash Player as vector graphics. These two engines manipulate the display object's position and scale based on perspective projection algorithms, instead of drawing a bitmap and apply it to a 3D object.

Stardust Supports Particle Masking

This feature was inspired by the collision filtering feature of the renowned Box2D Physics Engine. This feature enables you to assign an integer "mask" value to each particle and `Action` object; if the bitwise-AND value of a particle's and an action's mask values is zero, the particle's properties would not be manipulated by the action.

Stardust Supports Gravity And Deflector Simulation

The `Gravity` class can simulate gravity effects, using along with the `Field` class. You can use this feature to create effects like earth gravity, wind, black hole, and so on.

The `Deflect` class allows you to manipulate a particle's position and velocity as you like based on its current position and velocity. This class is used along with the `Deflector` class. There are already several built-in deflectors, such as bounding box, wrapping box, and solid obstacles.

Stardust Supports Action Triggers

The `ActionTrigger` class allows you to create even more complex particle behaviors. It's a conditional action: when certain conditions are satisfied, actions linked to the trigger will be used to manipulate particles.

A good example is combining the `DeathTrigger` and `Spawn` classes. Upon the death of a particle, new particles are spawned at the very position of the dead particle – this can be used to create firework effects.

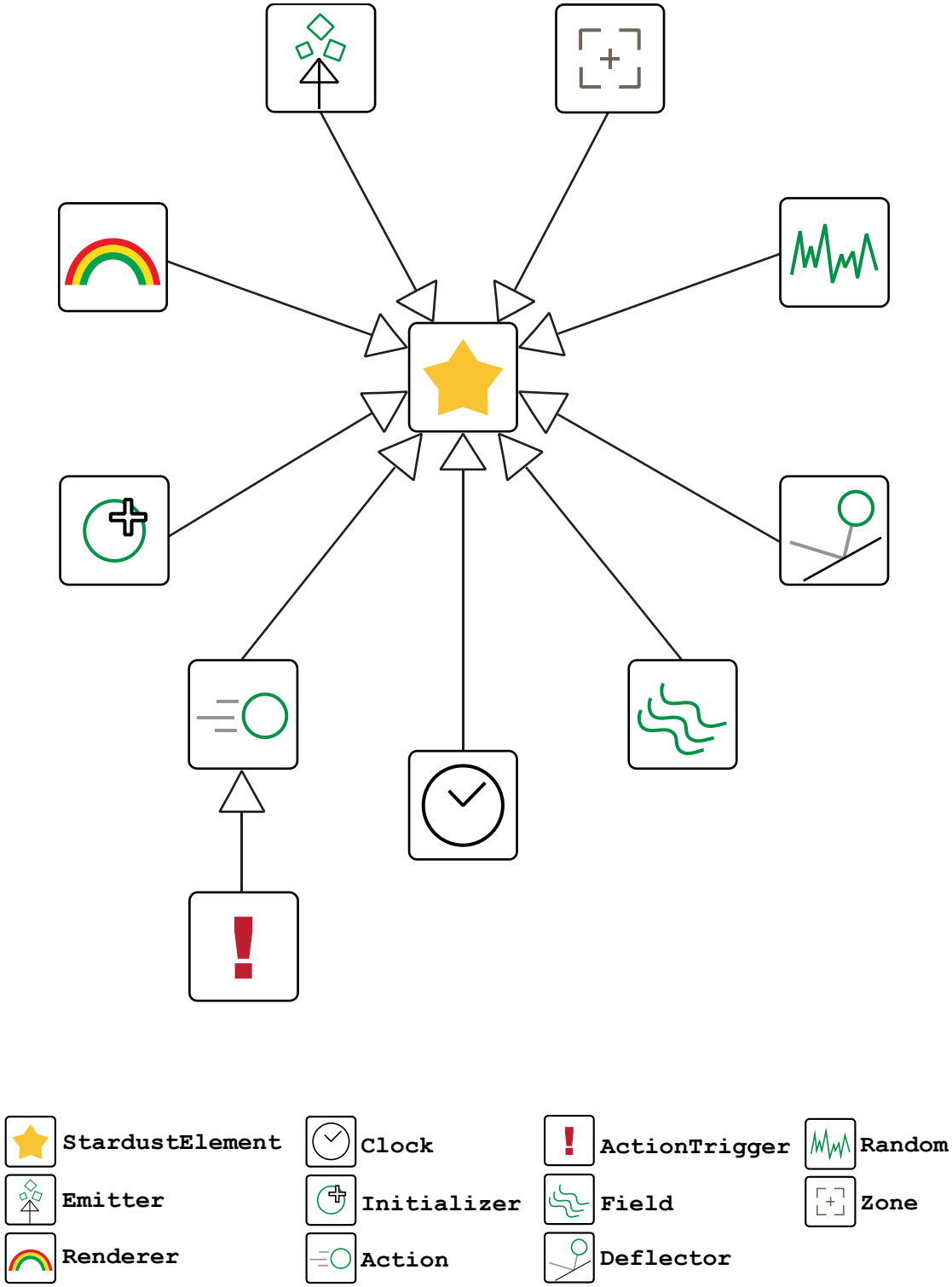
Stardust Supports XML

This is actually my favorite feature. Imagines this: your boss wants you to create a particle effect and you didn't let him down. After a couple of days he asks you to tweak several parameters; you do what he asks for and recompile the program. If the same scenario keeps recurring over and over, you'll be recompiling the codes to death. You can be sure about that, especially when your main program is a big one.

The XML feature allows you to separate the particle effect's parameters from the main application's source code. Stardust can generate an XML representation of a particle system (don't get confused with the synonym for a particle engine, by saying **particle system** here I mean a set of particle engine elements working together as a whole). You can dynamically load this XML representation at run-time and parse it using Stardust's built-in XML parser to reconstruct the exact same particle system.

After separating the particle system parameters from the main program as an external file, the story changes: no more recompilation is needed. All you need to do is edit your XML file using your favorite text editor, save it, rerun your main program, and you can wait to see your boss's happy face.

Class Inheritance Diagram



Class Responsibilities

You can check out the full API documentation here.

The `StardustElement` Class

This is the base class of the following classes: `Emitter`, `Initializer`, `Action`, `Clock`, `Renderer`, `Random`, `Field`, `Deflector`, and `Zone`.

This class defines the basic properties and methods of an element in a particle system (again, I'm referring to elements working together as a whole), such as element name and XML conversion.

The `Emitter` Class

This class is in charge of creating new particles, updating particles in the main loop, and removing dead particles from simulation. You have to repeatedly call its `step()` method to keep the simulation going. The `step()` method is the main loop function of a Stardust particle system.

The `Initializer` Class

This class initializes a particle's properties just once upon the particle's birth. An initializer can be added to an emitter through the emitter's `addInitializer()` method.

Initializers can initialize a particle's position, velocity, rotation, angular velocity, mass, mask, alpha, scale, and so on.

The `Action` Class

This class manipulates a particle's properties on each call of an emitter's `step()` method.

Actions can update a particle's position according to the particle's velocity, change a particle's velocity according to gravity fields or deflectors, and so on.

The `Clock` Class

This class decides how frequently an emitter creates new particles. Upon an emitter's `step()` method call, the emitter calls its associated clock's `getTicks()` method to determine how many new particles to create.

The `Renderer` Class

The emitter class updates particle data. The data is only numerical data in a computer's memory. This class is used to visualize the numerical data onto the screen.

You can extend this class to create your own renderer that works with specific 3D engines. Engine-specific codes are only written in this class and initializers.

The `Random` Class

This class is in charge of generating random values that can used to create the randomness of particles. Remember my own definition for particle effects? I said particles are entities with similar but not exactly same appearance and behaviors. In other words, particles possess a quality of randomness.

Many initializers make use of this class to create particles with random properties, such as the `Rotation`, `Scale`, and `Alpha` classes.

The `Field` Class

This class represents vector fields. Vector fields can be used as gravity fields, impulse fields, etc.

For example, a `UniformField` object, which is, as its name suggests, a uniform vector field, can be used to create earth gravity if pointed downward, or create wind effect if the `massless` property is set to false.

The `Deflector` Class

This class takes a particle's position and velocity as inputs and outputs a new position and velocity. Used with the `Deflect` action, you can manipulate a particle's position and velocity based on its original position and velocity.

The `Zone` Class

This class represents a geometric zone. A zone generates a random point contained in the zone through the `getPoint()` method. This point can be used to initialize a particle's position or velocity.

The 3D version of the `Deflector`, `Field`, and `Zone` classes are the `Deflector3D`, `Field3D`, and `Zone3D` classes, respectively.

The following classes are not subclasses of the `StardustElement` class. They are value object (VO) classes.

The `Particle` Class

This class is the VO class for particles. It defines particle properties, such as mask, mass, scale, alpha, and collision radius. The two subclasses, `Particel2D` and `Particle3D`, further define 2D- and 3D-specific properties, such as position and velocity.

The `Vec2D` And `Vec3D` Classes

These classes represent vectors in 2D and 3D. They provide methods for common vector operations, such as dot product and cross product.

The `MotionData2D` and `MotionData3D` Classes

These classes are degenerated versions of the vector classes: their sole purpose is to store values, without providing any methods for vector operations.

The `MotionData4D` and `MotionData6D` Classes

These two classes hold double the number of properties as the `MotionData2D` and `MotionData3D` classes. They are mainly used by the deflectors for holding both position and velocity information of particles.

General Workflow

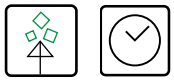
Overview

The following are the steps for the general use of Stardust:

1. Create a clock.
2. Create an emitter, and set the emitter's `clock` property to the clock created in the previous step.
3. Create a renderer.
4. Add the emitter to the renderer through the renderer's `addEmitter()` method.
5. Add initializers to the emitter through the emitter's `addInitializer()` method.
6. Add actions to the emitter through the emitter's `addAction()` method.
7. Listen to the `Event.ENTER_FRAME` or `TimerEvent.TIMER` event to repeatedly call the emitter's `step()` method, which is the main loop of the engine.

The Main Loop

The main loop (the `Emitter.step()` method call) involves the following process:



1. Clock Checking Phase

The emitter asks the clock how many new particles it should create. The return value of the clock's `getTicks()` method determines the number of particles to create.

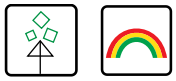
2. New Particle Creation Phase

The emitter creates new particles of the number determined by the clock, and adds the new particles to its particle list. And the emitter asks its initializers to initialize the new particles.

3. New Particle Initialization Phase

The new particles are initialized by the initializers through their `initialize()` methods.

Note that each initializer has a `priority` property; the initializers are sorted according to this property and used to initialize new particles in the sorted order.



4. New Particle Rendering Phase

When the new particles are added and initialized, the emitter dispatches an `EmitterEvent.PARTICLES_ADDED` event. The renderer listens for this event, and the event invokes the renderer's `particlesCreated()` method. This method is in charge of engine-specific newly created particle handling.

For instance, the `DisplayObjectRendrerer` adds the new particles' corresponding display objects to a display object container's display list.

5. Action Preupdate Phase

The emitter calls the `preupdate()` methods of all its actions to set up the actions before actually updating particles.

For instance, the `Collide` action calculates the `maxDistance` value during this phase to optimize the upcoming collision calculation.

Note that an action may have a `preupdate()` method that does nothing, which is in fact true for most actions.

6. Action Phase

During this phase, the emitter asks all its actions to update the particles in the emitter's particle list through the actions' `update()` method.

Note that each action has a `priority` property; the actions are sorted according to this property and used to update particles in the sorted order.

Also, each action has a `mask` property, the action would ignore and not update a particle if the bitwise-AND of the particle's `mask` property and the action's `mask` property is zero. This is useful for masking out some particles for specific actions. A particle's `mask` value can be initialized by the `Mask` initializer.

7. Action Postupdate Phase

The emitter calls the `postupdate()` methods of all its actions to handle the end of the update.

Note that an action may have a `postupdate ()` method that does nothing, which is in fact true for most actions.

8. Dead Particle Removal Phase

The emitter checks the `isDead` property of its particles to see if there are dead particles.

9. Dead Particle Rendering Phase

All dead particles are removed from the emitter's particle list. And the emitter dispatches an `EmitterEvent.PARTICLES_REMOVED` event. The renderer listens for this event, and the event invokes the renderer's `paritclesRemoved()` method. This method is in charge of engine-specific dead particle removal handling.

For instance, the `DisplayObjectRenderer` removes the dead particles' corresponding display objects to a display object container's display list.

10. Live Particle Rendering Phase

Before the main loop ends the emitter dispatches one last event: the `EmitterEvent.STEPPED` event. This event invokes the renderer's `render()` method, which is in charge of engine-specific rendering.

For instance, the `DisplayObjectRenderer` sets the `x`, `y`, `rotation`, `scaleX`, `scaleY`, and other `DisplayObject` properties of the live paritcles' corresponding display objects, based on the numeric data stored in the emitter's particle list.

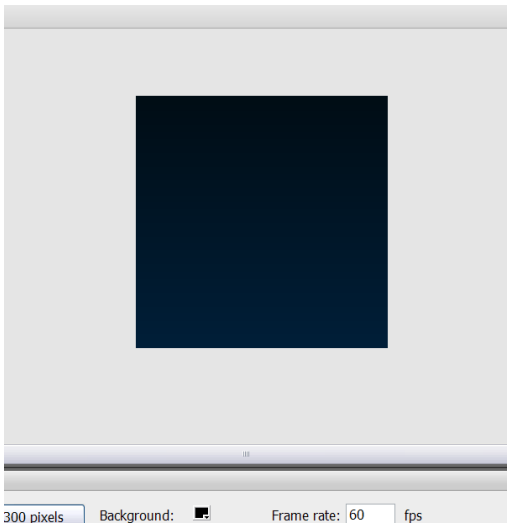
A Minimalistic Example: Rain

Step by Step

Let's go through a minimalistic example to help you get started with Stardust. We are going to create a simple raining effect.

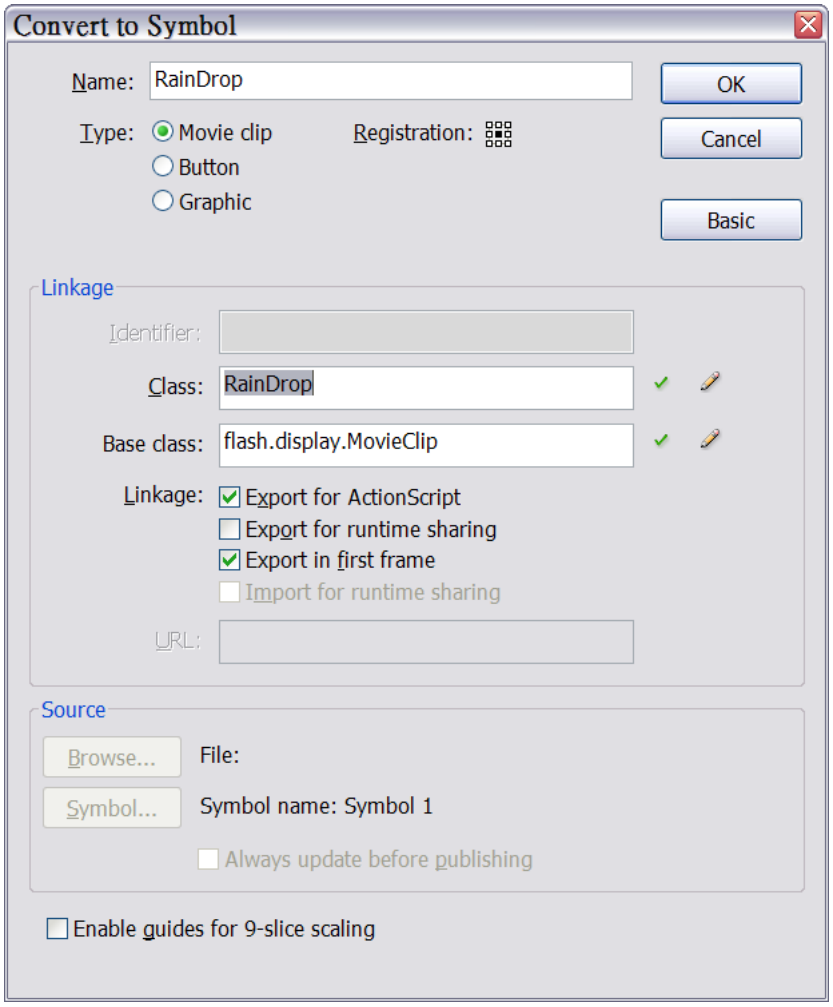
1. Open the Flash IDE and create a new FLA file.
   Stardust works with both Flash CS3 and CS4, so you can choose whichever version you like. Here I'm using Flash CS3.

2. We are going to create a raining effect at night, so create a dark background for your Flash document.



3. Draw a rain drop.

4. Convert the rain drop to a `MovieClip` symbol. Export it for ActionScript and name it `RainDrop`. Then you can delete the instance on the stage.



5. Now let's get down to some coding. Open the Actions panel and make sure the first frame is selected. If you accidentally create more than one frame, delete the other frames. Having more than one frame will result in unwanted errors.

6. Recall the classes from the previous chapter. We'll need an emitter, a renderer, a clock, some initializers, a couple of actions, and a few `Random` objects as well as `Zone` objects. The objects we're going to use consist of common ones for both 2D and 3D, as well as some 2D-specific ones. We'll have to import the following packages. Common objects are inside the `idv.cjcat.stardust.common` package, while the 2D and 3D ones are inside the `idv.cjcat.stardust.twoD` package and the `idv.cjcat.stardust.threeD` package, respectively.

7. Import the required packages.

```
import idv.cjcat.stardust.common.actions.*;
import idv.cjcat.stardust.common.clocks.*;
import idv.cjcat.stardust.common.initializers.*;
import idv.cjcat.stardust.common.math.*;
import idv.cjcat.stardust.twoD.actions.*;
import idv.cjcat.stardust.twoD.emitters.*;
import idv.cjcat.stardust.twoD.initializers.*;
import idv.cjcat.stardust.twoD.renderers.*;
import idv.cjcat.stardust.twoD.zones.*;
```

8. Create an emitter, and create a steady clock for the first constructor parameter. This clock will be the clock associated with the emitter. Pass a number 1 as the steady clock's constructor parameter. This causes the emitter to create one new particle on each main loop call. Since this is a 2D example, we're going to use the `Emitter2D` class, which is a subclass of the `Emitter` class for 2D particle effects.

```
var emitter:Emitter2D
      = new Emitter2D(new SteadyClock(1));
```

9. No we're going to create a renderer which renders display objects onto the screen. The `DisplayObjectRenderer` class is for such purpose. Create a `Sprite` object and add it to the display list. Pass this sprite as the renderer's constructor parameter. This tells the renderer to add new particles to and remove dead particles from the sprite.

```
var sprite:Sprite = new Sprite();
addChild(sprite);

var renderer:DisplayObjectRenderer
      = new DisplayObjectRenderer(sprite);

renderer.addEmitter(emitter);
```

10. Now add initializers to the emitter to initialize our raindrop particles. The `DisplayObjectClass` initializer class tells the emitter to create a RainDrop instance for each particle. The `Position` initializer class initializes particle positions based on a `Line` zone object with two end points at (0, 0) and (0, 300).

    The `Velocity` object initializes particle velocities based on a `SinglePoint` zone object. Velocities are set based on zones: the zone object yields a coordinate of a random point in the zone, and the vector pointing from the origin (0, 0) to the random point is then assigned to the particle by the `Vector` initializer class as the particle's initial velocity.

```
var displayObjectClass:DisplayObjectClass
      = new DisplayObjectClass(RainDrop);
var position:Position
      = new Position(new Line(0, 0, 300, 0));
var velocity:Velocity
      = new Velocity(new SinglePoint(0, 5));

emitter.addInitializer(displayObjectClass);
emitter.addInitializer(position);
emitter.addInitializer(velocity);
```

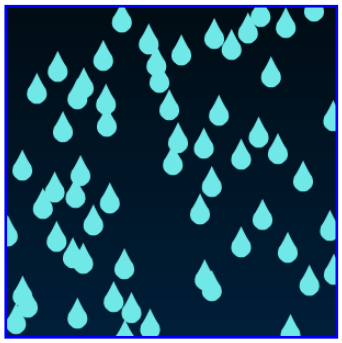
11. Add actions to the emitter. The `Move` action object causes particles to move according to their velocities. The `DeathZone` action object causes particles to die when they get beyond a given zone, which is a `RectZone` zone object in this example. Dead particles are removed from the display list by the renderer.

```
var move:Move = new Move();
var deathZone:DeathZone
    = new DeathZone(new RectZone(0, 0, 300, 300), true);

emitter.addAction(move);
emitter.addAction(deathZone);
```

12. Finally, repeatedly call the emitter's `step()` method to keep the particle effect going. Simply listen to the `Event.ENTER_FRAME` event.

```
addEventListener(Event.ENTER_FRAME, emitter.step);
```

13. Compile and test the movie. If all goes right, you should see the raining effect working properly.



14. Now it's time to add some randomness to the rain drops. We're going to add a random drift in the X direction. So we'll add an `AbsoluteDrift` action object to the emitter. A `UniformRandom` random object is used to generate the drift amount. The first constructor parameter is set to 0 and second 0.1; this means the random object generates a random number ranging from -0.1 to 0.1 (0.1 of variation centered at 0). Moreover, we'll add an `Oriented` action object to the emitter, so that the rain drops will align to their velocity direction. Note that the `offset` property of the action is set to 180. It's because for Stardust, a symbol's up direction is considered the front of a particle by default. The front of the rain drop symbol is apparently down, so we need to give it an offset of 180 degrees to fix this.

```
var drift:RandomDrift = new RandomDrift();
drift.randomX = new UniformRandom(0.1, 0);
var oriented:Oriented = new Oriented();
oriented.offset = 180;

emitter.addAction(drift);
emitter.addAction(oriented);
```

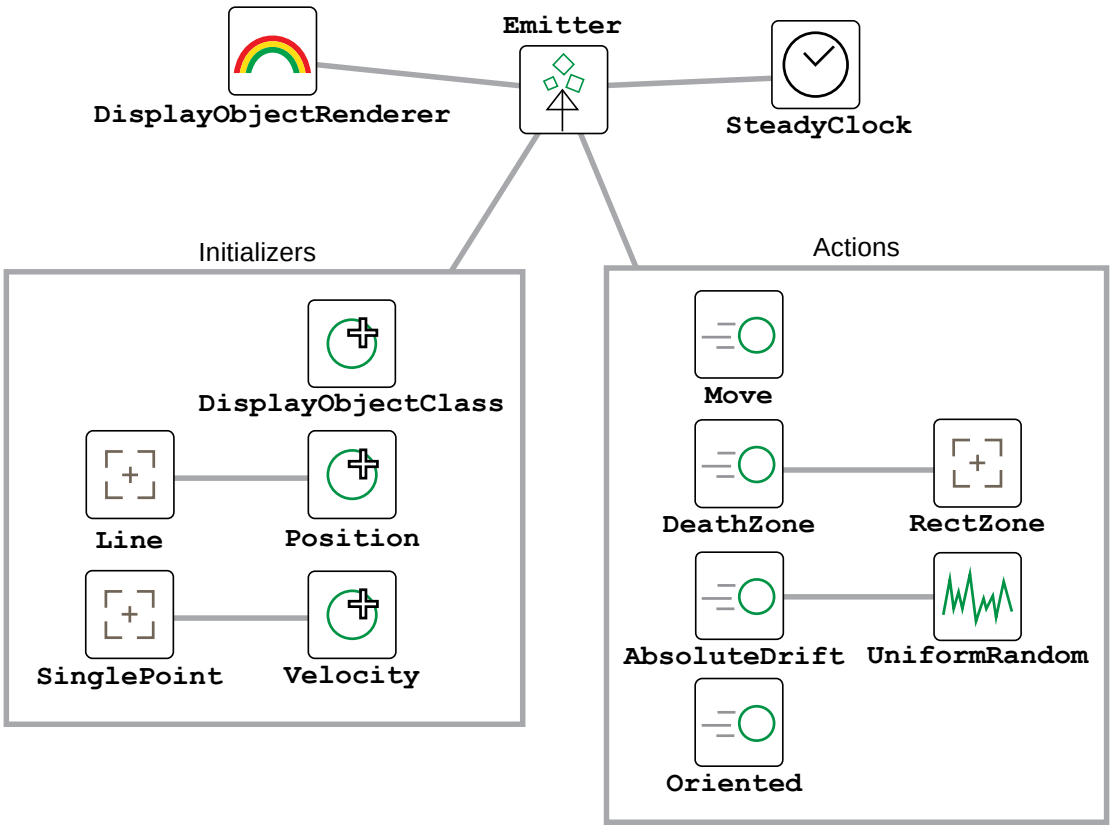
15. Compile and test the movie. Now you can see random drift is applied to particles in the X direction, and particles align to their velocity directions.

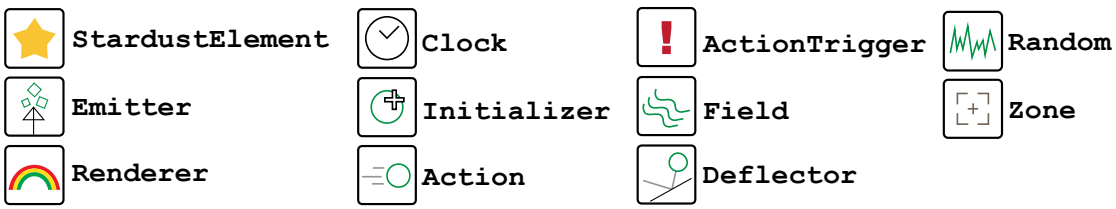
Behind The Scene

Let's take a detailed look into what the example is all about.

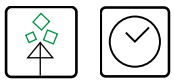
Here are the relations among the objects:

An emitter is added to a renderer.
A clock is linked to the emitter.
Several initializers are added to the emitter.
Several actions are added to the emitter.





Remember **the main loop** described in the previous chapter? Let's see what each phase of the loop exactly does in this example:



1. Clock Checking Phase

The emitter asks the clock how many new particles it should create. The steady clock's `getTicks()` method returns 1, meaning that the emitter should create one new particle.

2. New Particle Creation Phase

The emitter creates one new particle and adds the particle to its particle list. The emitter then asks the initializers to initialize this new particle.

3. New Particle Initialization Phase

The `DisplayObjectClass` initializer sets the particle's target to a new display object.

The `Position` initializer sets the particle's position based on a random point on the Line zone object.

The `Velocity` initializer sets the particle's position based on the `SinglePoint` zone object, which only returns a single point.


4. New Particle Rendering Phase

The renderer adds the new display object corresponding to the new particles to the display list.

5. Action Preupdate Phase

Every action in this example does nothing in the `preupdate()` method.

6. Action Phase

The `Move` action updates the particles position based on their own velocities.

The `DeathZone` action checks if a particle exceeds the `RectZone` zone. If yes, the particle is marked as dead.

The `AbsoluteDrift` action applies a random drift in the X direction to the particles based on the `UniformRandom` random object.

The `Oriented` action aligns the particles to their velocities.

7. Action Postupdate Phase

Every action in this example does nothing in the `postupdate()` method.

8. Dead Particle Removal Phase

The emitter checks the `isDead` property of its particles to see if there are dead particles.

9. Dead Particle Rendering Phase

The renderer removes dead particles' display objects from the display list.

10. Live Particle Rendering Phase

The renderer updates all the remaining display objects' positions, rotation, scale, alpha, etc.

This is the end of this minimalistic example. I hope this example helps you understand how Stardust works.

Detailed Look into Some Stardust Elements

In this chapter, I'll go through some Stardust elements that are worth mentioning.

Initializers

The `CompositeInitializer` Class

You can add multiple initializers to a composite initializer through its `addInitializer()` method, and the composite initializer acts like a single initializer. A composite initializer's `initialize()` method applies each component initializer's `initialize()` method to the particles to be initialized.

This class is usually used with the `SwitchInitializer` class, to cause an emitter to use different set of initializers to initialize new particles.

The `SwitchInitializer` Class

This initializer chooses among multiple initializers to initialize particles. When a particle is to be initialized by a switch initializer, the initializer picks just one initializer out of its collection and uses it to initialize the particle. You can also specify a "weight" of a initializer for the switch initializer. An initializer with higher weight is more likely to be chosen.

Actions

The `CompositeAction` Class

Like the `CompositeInitializer` class, the `CompositeAction` class is a group of actions. By default, when its `update()` method is called to update particles, it calls all the underlying component actions' `update()` method to update the particles, without further checking the mask.

If the `checkComponentMasks` property, `false` by default, is set to `true`, each component action's mask is tested with the particles' mask property (see **The Action Class** section in the **Class Responsibilities** chapter).

The `AlphaCurve` Class

You can use this class to create effects such as particles fading from transparent to opaque upon birth and fading back to transparent near death. The `inLifespane` and `outLifespane` properties determine the fading time. The fading curve (or fading function) is the Linear.easeIn function by default. You can use other easing functions created by Robert Penner for the fading curve, by assigning easing function references to the `inFunction` or `outFunction` property.

The `inAlpha` property is the initial alpha value and the `outAlpha` final.

The `ScaleCurve` Class

Similar to the `AlphaCurve` class, only that this class works on the scales of particles instead of alpha values.

The `inScale` property is the initial scale and the `outScale` final.

The `Spawn` Class

(3D version: `Spawn3D`)

This action spawns new particles right at the position of an existing particle. The `inheritVelocity` property determines if the spawned particles inherit the original particle's velocity. The `inheritDirection` property determines if the spawned particles' velocities are rotated for an angle equal to the angle between the original particle's velocity direction and the Y axis.

The newly spawned particles are initialized by initializers added to a `Spawn` action through the `addInitializer()` method.

The `Collide` Class

(3D version:`Collide3D`)

This action causes particles to collide against each other. Two particles will only collide if their masks have a nonzero bitwise-AND value.

Action Triggers

The `DeathTrigger` Class

This trigger is triggered when a particle's life reaches zero. Used with the `Spawn` action, you can create a fireworks effect.

The `ZoneTrigger` Class

(3D version: `ZoneTrigger3D`)

This trigger is triggered when a particle is contained by a give zone. Used with the Damping action, you can create an effect in which particles decelerate when contained in a certain zone.

Zones

The `CompositeZone` Class

This class represents a group of zone. You can add zones to this group through the `addZone()` method. When a random point is requested from a composite zone, it first randomly chooses a component zone and gets a random point from the chosen zone. The larger the `area` property of a zone, the more likely the zone will be chosen.

The `Contour` Class

(3D version: `Surface`)

This class represents zones with zero thickness, such as points, lines, and curves. Its area property is actually the product of the length of the zone and its `virtualThickness` property (the `virtualThickness` property for `Surface` classes).

Working with XML

You already know the benefits that follow the use of external XML files from the **Feature Overview** chapter. Now let's take a closer look at the XML support feature.

The `XMLBuilder` Class

The `XMLBuilder` class does two things: serializing a particle system into XML representation and deserializing XML data to reconstruct the original particle system.

The static method `XMLBuilder.buildXML(element:StardustElement):XML` generates an XML object representing the particle system associated with the `element` parameter, which is a `StardustElement` object.

The XML representation comes in the following form:
(Some attributes are omitted for clarity)

```
<StardustParticleSystem version="1.0.XXX">
  <actions>
    <Age name="age"/>
    <DeathZone name="deathZone" zone="rectZone"/>
    <Move name="move"/>
  </actions>
  <clocks>
    <SteadyClock name="clock"/>
  </clocks>
  <emitters>
    <Emitter2D name="emitter" clock="clock">
      <actions>
        <Age name="age"/>
          <DeathZone name="deathZone">
          <Move name="move"/>
      </actions>
```

(continued on the next page)

(continuing the previous page)

```
        <initializers>
          <Life name="life" random="lifeRandom"/>
          <Mass name="mass" random="massRandom"/>
        <initializers/>
      </Emitter2D>
    </emitters>
    <initializers>
      <Life name="life" random="lifeRandom"/>
      <Mass name="mass" random="massRandom"/>
    </initializers>
    <randoms>
      <UniformRandom name="lifeRandom"/>
      <AveragedRandom name="massRandom"/>
    </randoms>
    <renderers>
      <DisplayObjectRenderer name="renderer">
        <emitters>
          <Emitter2D name="emitter"/>
        </emitters>
      </DisplayObjectRenderer>
    </renderers>
    <zones>
      <RectZone name="rectZone" width="650" height="500"/>
    </zones>
  </StardustParticleSystem>
```

Alright, that XML representation looks pretty long and scary. Why don't we take a look at a cuter version?

```
<StardustParticleSystem version="1.0.XXX">
  <actions>
    
    Age (name="age")
    
    DeathZone (name="deathZone" zone="rectZone")
    
    Move (name="move")
  </actions>
  <clocks>
    
    SteadyClock (name="clock")
  </clocks>
  <emitters>
    
    Emitter2D (name="emitter" clock="clock")
  </emitters>
  <initializers>
    
    Life (name="life" random="lifeRandom")
    
    Mass (name="mass" random="massRandom")
  </initializers>
  <randoms>
    
    UniformRandom (name="lifeRandom")
    
    UniformRandom (name="massRandom")
  </randoms>
  <renderers>
    
    DisplayObjectRenderer (name="renderer")
  </renderers>
  <zones>
    
    RectZone (name="rectZone")
  </zones>

</StardustParticleSystem>
```

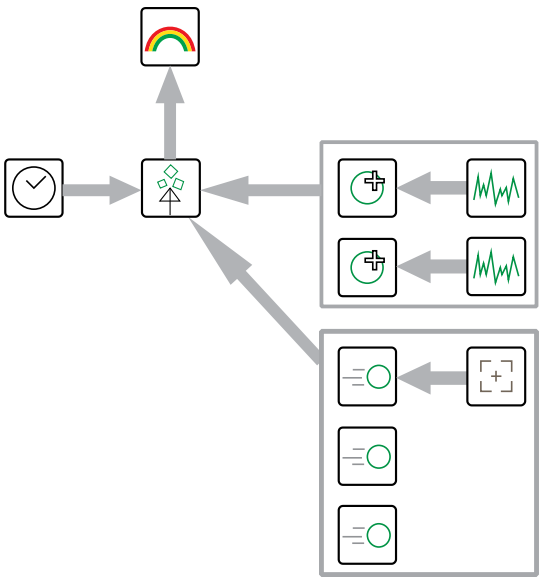
As you can see, the root node is the `<StardustParticleSystem/>` node, whose direct children nodes are parent nodes grouping together different types of elements. There are still other such nodes that are not present in this example: `<deflectors/>` and `<fields/>`.

For clarity, the `<Emitter2D/>` node's children nodes, `<actions/>` and `<initializers/>`, are omitted in the figure.

The relations in the original particle system are kept in this XML representation by using name references. For example, the `zone` attribute of the `<DeathZone/>` node indicates that the `DeathZone` action's `zone` property is a reference to a `Zone` object named `rectZone`, which can be found in the `<zones/>` node. You can check out other relations in the figure yourself by finding out matching attribute values and element names.

Related Objects

If you convert the figure into a tree-like structure, it looks like this:



The arrows represent the "has-a" relations: the renderer has an emitter, the emitter has a clock, the emitter has two actions, and so on.

You can see that the renderer is the root node of the tree. It's because the XML representation is generated with the renderer passed as the `element` parameter of the `XMLBuilder.buildXML()` method.

The `XMLBuilder` class determines that the emitter, clock, actions, initializers, random objects, and the zone are related to the renderer, the emitter having a direct relation to the renderer and the rest indirect.

To represent the entire particle system in respect to the renderer, the `XMLBuilder` class includes all the related objects in the final XML representation output, and uses names to refer to related objects.

How does the `XMLBuilder` class know what objects are related? The answer is the `StardustElement.getRelatedObjects()` method. This method returns an array of objects that are related to an element. By calling this method recursively, the `XMLBuilder` class then gets to know all the objects related to the element passed to the builder's `buildXML()` method.

This also means if you want to extend your own `StardustElement` class, you should override the `getRelatedObjects()` method and return the objects you think that must be included in the XML representation for future reconstruction.

The requirement of specifying related objects of your own custom classes yourself might seem troublesome at the first glance, but it has a good reason for that: by allowing you to decide which objects are included in the XML representation, you can hide internally used elements from the public, reducing unnecessarily excessive amount of information.

For instance, the `LazyAction` class is a `CompositeAction` subclass that internally includes many actions. By default, a composite action's XML representation lists out all the component actions. The purpose of the `LazyAction` class is to let developers tweak a few intuitive parameters to achieve complex actions. Instead of displaying all the underlying component actions in the XML representation, simply showing the easy-to-understand

parameters as XML attributes is more eye-friendly. So the `LazyAction` class overrides the `getRelatedObjects()` method and makes it return an empty array. The `toXML()` method is overridden to generate XML representations with easy-to-understand attributes, and the `parseXML()` method is overridden to parse them.

The toXML() and parseXML() Methods

The StardustElement class defines the `toXML()` and `parseXML()` methods. The former is for generating an element's XML representation for the `XMLBuilder` class to include in a particle system's complete XML representation. And the latter is for parsing the element's XML representation and reconstructing the original element's properties.

The `XMLBuilder.buildFromXML()` method.

The `XMLBuilder.buildFromXML(xml:XML):void` method reconstructs a particle system from its XML representation. Note that this is not a static method, so you have to instantiate an `XMLBuilder` object to use the method.

After a builder's `buildFromXML()` method is called for particle system reconstruction, the builder holds the entire reconstructed particle system within it. You can access the elements in the system by calling the builder's `getElementByName()` method.

For instance, after a builder reconstructed a particle system from the system's XML representation, if an `Emitter2D` object in system has a name "`myEmitter`", you can retrieve the reconstructed emitter's reference by writing the following code:

```
var e:Emitter2D =
    builder.getElementByName("myEmitter") as Emitter2D;
```

This concludes the XML feature of Stardust.

Example: XML Deserialization

In this example, I'm going to show you how to use the `XMLBuilder` class to deserialize existing XML data to reconstruct the underlying particle system.

1. Copy the following XML code and save it as an external file, named "`particleSystem.xml`".

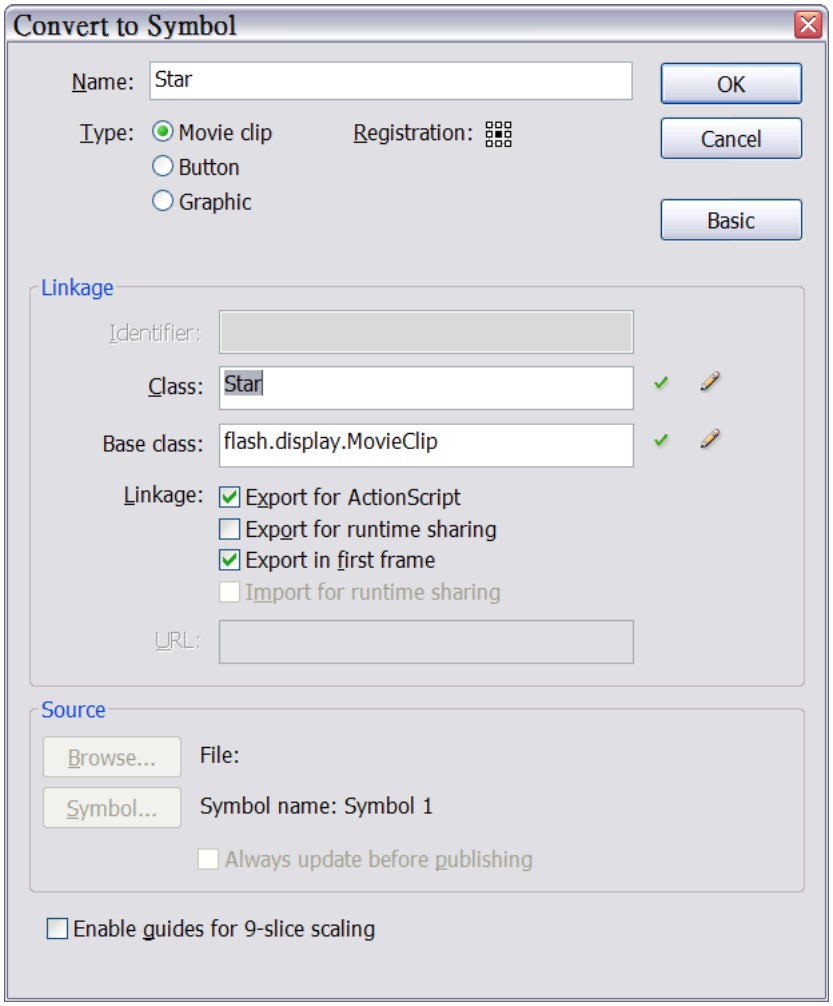
   (It's kind of messy and goes across two pages. Just copy it and paste.)

```
<StardustParticleSystem version="1.0.80 Beta">
  <actions>
    <Age name="Age_0" active="true" mask="1" multiplier="1"/>
    <DeathLife name="DeathLife_0" active="true" mask="1"/>
    <Move name="Move_0" active="true" mask="1" multiplier="1"/>
    <ScaleCurve name="ScaleCurve_0" active="true" mask="1"
inScale="0" outScale="0" inLifespan="0" outLifespan="10"
inFunction="Linear.easeIn" outFunction="Linear.easeOut"/>
  </actions>
  <clocks>
    <SteadyClock name="SteadyClock_0" ticksPerCall="1"/>
  </clocks>
  <emitters>
    <Emitter2D name="Emitter2D_0" clock="SteadyClock_0">
      <actions>
        <ScaleCurve name="ScaleCurve_0"/>
        <Age name="Age_0"/>
        <DeathLife name="DeathLife_0"/>
        <Move name="Move_0"/>
      </actions>
      <initializers>
        <DisplayObjectClass name="DisplayObjectClass_0"/>
        <Position name="Position_0"/>
        <Life name="Life_0"/>
        <Rotation name="Rotation_0"/>
        <Velocity name="Velocity_0"/>
      </initializers>
```

```
    </Emitter2D>
  </emitters>
  <initializers>
    <DisplayObjectClass name="DisplayObjectClass_0"
active="true"/>
    <Life name="Life_0" active="true" random="UniformRandom_0"/>
    <Position name="Position_0" active="true"
zone="SinglePoint_0"/>
    <Rotation name="Rotation_0" active="true"
random="UniformRandom_1"/>
    <Velocity name="Velocity_0" active="true"
zone="LazySectorZone_0"/>
  </initializers>
  <randoms>
    <UniformRandom name="UniformRandom_0" center="30"
radius="5"/>
    <UniformRandom name="UniformRandom_1" center="0"
radius="180"/>
  </randoms>
  <renderers>
    <DisplayObjectRenderer name="DisplayObjectRenderer_0"
addChildMode="0" forceParentChange="false">
      <emitters>
        <Emitter2D name="Emitter2D_0"/>
      </emitters>
    </DisplayObjectRenderer>
  </renderers>
  <zones>
    <LazySectorZone name="LazySectorZone_0" rotation="0" x="0"
y="0" minRadius="2" maxRadius="4" minAngle="-270" maxAngle="90"
radius="3" radiusVar="1" directionX="0" directionY="-1"
directionVar="180" />
    <SinglePoint name="SinglePoint_0" rotation="0"
virtualThickness="1" x="320" y="240"/>
  </zones>
</StardustParticleSystem>
```

2. Open the Flash IDE and create a new FLA file, and draw a star.

3. Convert the star to a MovieClip symbol, and export it for ActionScript with a class name `Star`.



4. Import the required packages.

```
import idv.cjcat.stardust.common.actions.*;
import idv.cjcat.stardust.common.clocks.*;
import idv.cjcat.stardust.common.initializers.*;
import idv.cjcat.stardust.common.math.*;
import idv.cjcat.stardust.common.xml.*;
import idv.cjcat.stardust.twoD.actions.*;
import idv.cjcat.stardust.twoD.emitters.*;
import idv.cjcat.stardust.twoD.initializers.*;
import idv.cjcat.stardust.twoD.renderers.*;
import idv.cjcat.stardust.twoD.zones.*;
```

5. Create three variables for holding retrieved element references.

```
var builder:XMLBuilder = new XMLBuilder();
var emitter:Emitter2D;
var singlePoint:SinglePoint;
```

6. Now we need to register a mapping between an XML tag name and an actual class, which would cause the compiler to include these classes during compilation. This allows the builder to find the right classes during run-time; otherwise, the builder cannot find the correct constructor to invoke, and would throw a run-time error.

   We have two alternatives to register classes to the builder. The first one is registering classes one by one through the `registerClass()` method; the builder would call the `getXMLTagName()` method of an instance of the registered class, and remember its returned string. Later on, if the builder parses the XML data and encounters an XML tag with the same name, it will know the tag refers to this class.

The following code does just that:

```
builder.registerClass(Age);
builder.registerClass(DeathLife);
builder.registerClass(Move);
builder.registerClass(ScaleCurve);
builder.registerClass(SteadyClock);
builder.registerClass(Emitter2D);
builder.registerClass(DisplayObjectClass);
builder.registerClass(Position);
builder.registerClass(Life);
builder.registerClass(Rotation);
builder.registerClass(Velocity);
builder.registerClass(UniformRandom);
builder.registerClass(DisplayObjectRenderer);
builder.registerClass(LazySectorZone);
builder.registerClass(SinglePoint);
```

That's a lot of lines. If you would like to include a group of classes at once, you can use the `ClassPackage` subclasses, which is essentially a class group, and the builder's `registerClasseFromClassPackage()` method.

The following code is an alternative:

```
builder.registerClassesFromClassPackage
                    (CommonClassPackage.getInstance());
builder.registerClassesFromClassPackage
                    (TwoDClassPackage.getInstance());
```

The code is much shorter and cleaner. But please beware: the second alternative actually includes all the classes in the `common` and `twoD` packages, greatly increasing the file size of the final compiled program. If you are working on a file-size-sensitive project, such as online banner, I would recommend you use the first alternative, listing just the classes you want to use.

7. Load the external XML file "`particleSystem.xml`".

```
var loader:URLLoader = new URLLoader();
loader.addEventListener(Event.COMPLETE, loaded);
loader.load(new URLRequest("particleSystem.xml"));
```

8. Write the definition for the `loaded` listener. The XML representation only contains information related to the engine, independent of the original context in which it was being generated; which means the `DisplayObjectClass` initializer has a `null` value for its `displayObjectClass` property, and the `container` property of the `DisplayObjectRenderer` object is also `null`. We have to reset them in the `loaded` listener.

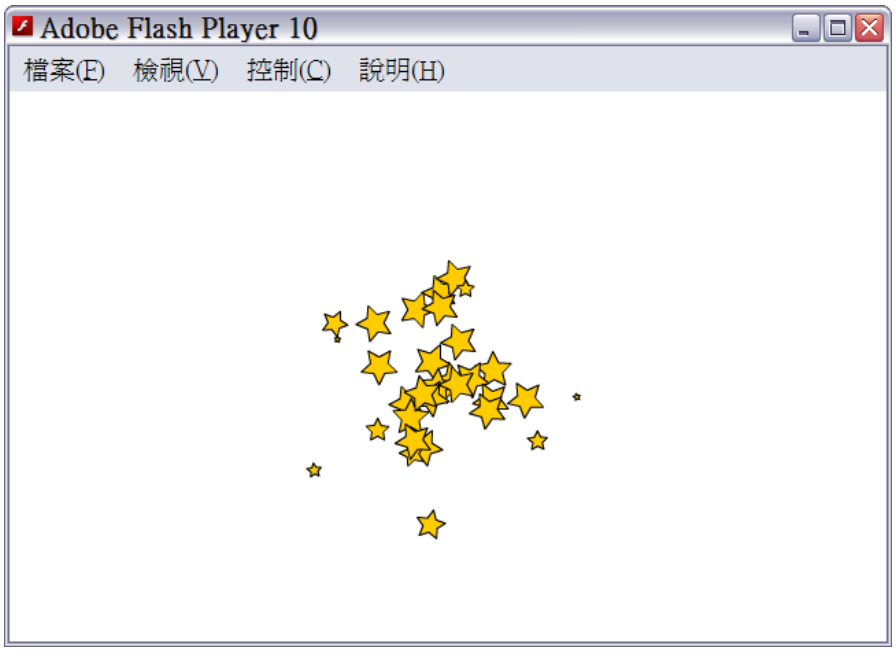
```
function loaded(e:Event):void {
    builder.buildFromXML(XML(loader.data));

    var displayClass:DisplayObjectClass =
    builder.getElementByName("DisplayObjectClass_0")
    as DisplayObjectClass;

    displayClass.displayObjectClass = Star;

    var renderer:DisplayObjectRenderer
    = builder.getElementByName("DisplayObjectRenderer_0")
    as DisplayObjectRenderer;

    renderer.container = this;

    emitter = builder.getElementByName("Emitter2D_0")
    as Emitter2D;

    singlePoint
    = builder.getElementByName("SinglePoint_0")
    as SinglePoint;
```

```
    //the main loop
    addEventListener(Event.ENTER_FRAME, loop);
}
```

9. Now it's time to write the main loop. We're going to make the `SinglePoint` zone used by the `Position` initializer follow the mouse cursor.

```
function loop(e:Event):void {
    singlePoint.x = mouseX;
    singlePoint.y = mouseY;
    emitter.step();
}
```

10. Now test the movie, and you can see stars shooting out from the mouse cursor and shrink as they die, caused by the `ScaleCurve` action.



11. To demonstrate the power of external XML files. Now open the "partycleSystem.xml" file and change line 37

    from

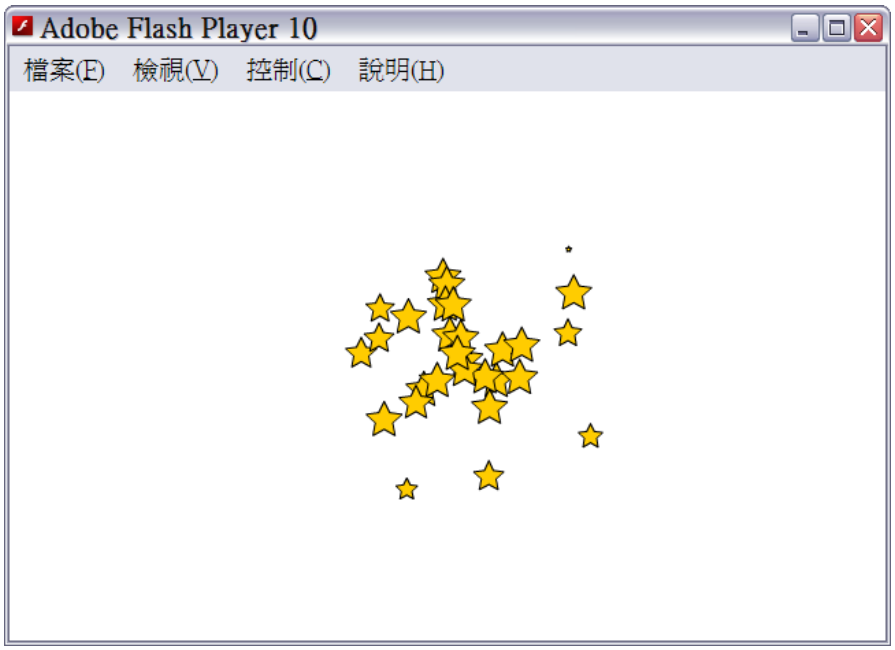
```
<UniformRandom name="UniformRandom_1" center="0" radius="180"/>
```

    to

```
<UniformRandom name="UniformRandom_1" center="0" radius="0"/>
```

12. Save the XML file and close it. Reopen the SWF file without recompiling the FLA file. And you'll notice that all stars now have zero rotation value.

    This is because the `Rotation` initializer uses the `UniformRandom` object corresponding to the XML tag we modified in step 10 to initialize particle rotation values. What we did essentially changed the random number rage from [-180, 180] to [0, 0].



Beyond This Manual

This is the end of this manual. I hope you have acquired a much better understanding of how to work with Stardust and how Stardust works internally. This manual does not cover every single feature provided by Stardust. There are much more for you to discover. You can check out the Stardust examples; also, you can look up in the documentation for more details on other actions, initializers, etc.

If you have any question, please feel free to ask me. My e-mail address is cjcat2266@gmail.com. There's also a forum for you to discuss Stardust with other users. Any feedback is highly appreciated.

Again, thank you very much for trying Stardust Particle Engine. This really means a lot to me.

Links

Stardust Project Homepage
http://code.google.com/p/stardust-particle-engine

Stardust Documentation
http://stardust-particle-engine.googlecode.com/svn/trunk/docs/index.html

Stardust Examples
http://stardust-particle-engine.googlecode.com/svn/trunk/examples/

Stardust Forum
http://groups.google.com/group/stardust-particle-engine

Stardust Project Donation
Click here to donate via Paypal

CJ's Blog
http://cjcat.blogspot.com