

Particle Universe Tutorial

Author: Henry van Merode
Version 0.1
www.fexpression.com

1	Introduction	3
2	A basic Particle System.....	4
2.1	Scripts and templates.....	4
2.2	Creation	4
2.3	Dynamic attributes	6
2.4	Add some movement.....	7
2.5	Colour your particles	8
2.6	Observing and event handling.....	10
2.7	Basic Collision	11
2.7.1	Colliders en Externs	11
2.7.2	Inter particle collision.....	12
2.7.3	Collision observer	13
3	Advanced options	15
3.1	Multiple techniques and renderers	15
3.2	Particle System LOD.....	15
3.2.1	Discrete LOD	15
3.2.2	Ramp function	16
3.3	Emission of non-visual particles	16
3.4	Atlas texture and TextureAnimator (V1.0 only)	17
3.5	Using PhysX (V1.0 only)	18
3.5.1	Fluid	20
4	Complex particle systems.....	22
4.1	Flocking behaviour.....	22
4.2	Flip Flop	22
4.3	Explosion.....	23

1 Introduction

Although the HTML documentation in the Particle Universe package describes the setup and scripting capabilities of Particle Universe, I decided to create an additional tutorial that gives you a step-by-step explanation of how to create a particle system. This tutorial is to compensate for the lack of an intuitive (official) Particle Universe Editor; creation of a particle system is very artist unfriendly and particle scripts can become very complex. The tutorial starts with the creation of some basic particle systems and covers more advanced features in later chapters.

This tutorial covers Particle Universe versions 0.8 and 0.81 with some new features from version 1.0.

2 A basic Particle System

2.1 Scripts and templates

Assuming you have compiled *Ogre*, the *Particle Universe plugin*, the *ParticleUniverseViewer* and have added the appropriate entries to the files *Plugins.cfg* and *resources.cfg* you are ready to start. If you haven't set it up yet, take a look at the *Setup.html* page in the Particle Universe manual (which can be found under `../ParticleUniverse/Docs/manual/`).

The easiest way (currently) to create a particle system is by means of a script. A Particle Universe script is identified by the extension *.pu*

If the directory in which your *.pu* scripts are saved is added to the *resources.cfg* file, all *.pu* scripts are parsed as soon as the Particle Universe plugin is loaded. During parsing of the scripts, particle system templates are created. These templates cannot be used themselves, but act as a blueprint for creation of your own particle system.

Besides *.pu* scripts you can also create *.pua* scripts. These scripts contain aliases and are parsed before the *.pu* scripts are parsed. An alias is a piece of script that can be used in other scripts (see also the manual for some more information).

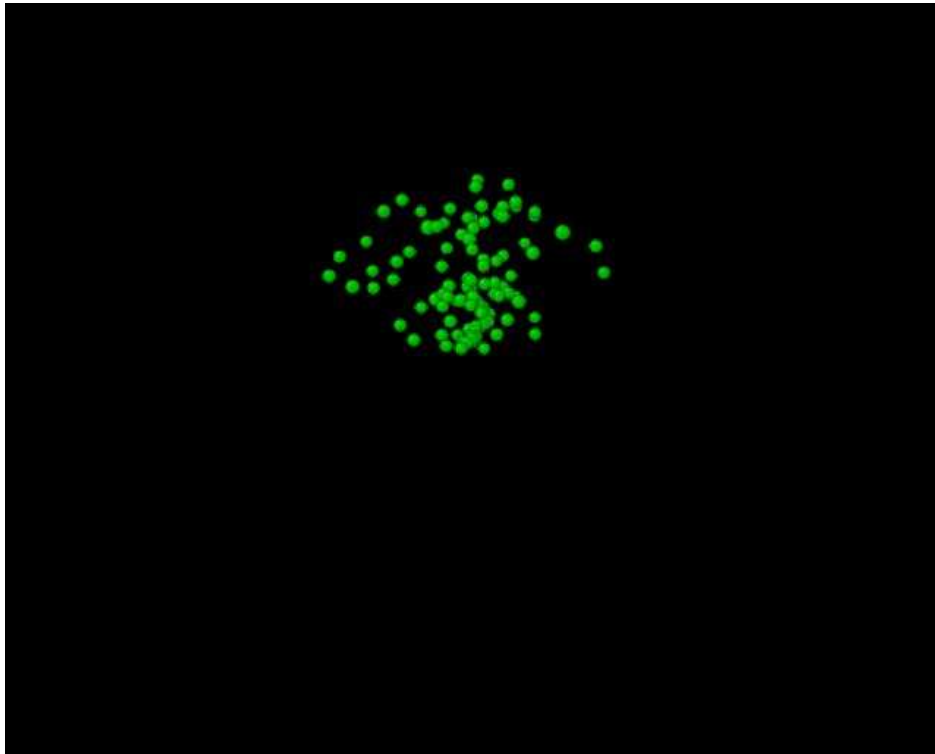
2.2 Creation

Use your favorite editor (Notepad is sufficient) to create a file called *tutorial.pu*. This file is saved in one of the directories that is included in *resources.cfg*. Add the following script to the *tutorial.pu* file.

```
system tutorial.2.2
{
    technique
    {
        renderer                Billboard {}
        material                 ParticleUniverse/GreenBall
        default_particle_width   20
        default_particle_height  20
        visual_particle_quota    100

        emitter                  Point
        {
            emission_rate        40
            angle                 60
            direction              0 1 0
            velocity               100
            time_to_live           3
        }
    }
}
```

Start the *ParticleUniverseViewer* application and search for tutorial.2.2 in the list. Select it and press the Start button. You should see something like this:



You have created a particle system that contains one technique (techniques are explained later) and this technique contains a renderer (always one) and one emitter. The emitter emits the particles and the renderer renders them. Look for a description of each attribute in the manual ([../ParticleUniverse/Docs/manual/index.html](http://ParticleUniverse/Docs/manual/index.html)) to see what it does.

You may have noticed in the manual that there are different types of renderers and emitters. They can be freely mixed, independent from each other. Change the following lines (in red):

```
system tutorial.2.2
{
  technique
  {
    renderer           Box {}
    ...
    emitter           Box
    {
      ...
    }
  }
}
```

This results in rendering of Boxes instead of Billboards and the particles don't emit from a point anymore, but are emitted from a (non-visible) Box with default dimensions (100 * 100 * 100).

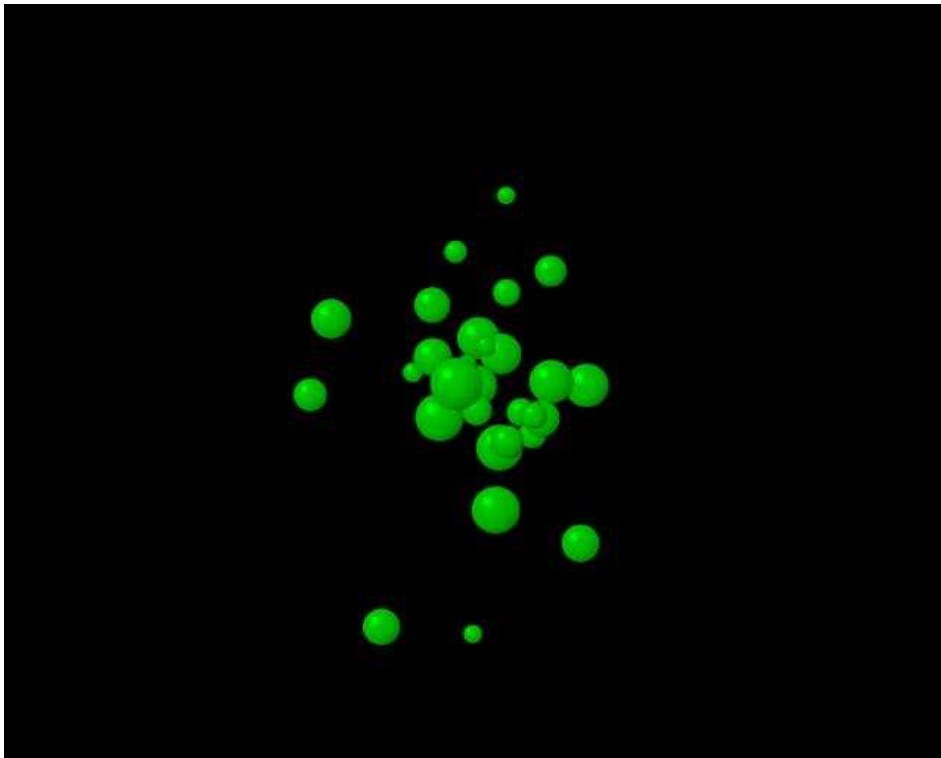
2.3 Dynamic attributes

A dynamic attribute is an attribute (property) in the script that can have a fixed value, a random value or a value that is generated by a function (sine, ...) or a curve (spline). This means that these types of attributes can be very dynamic because their value changes over time. Add the following example to your *tutorial.pu* script.

```
system tutorial.2.3
{
  technique
  {
    renderer          Billboard {}
    material          ParticleUniverse/GreenBall
    visual_particle_quota 500

    emitter          Point
    {
      emission_rate  10
      all_particle_dimensions
      {
        min          20
        max          80
      }
      angle          360
      direction      0 1 0
      velocity      dyn_random
      {
        min          40
        max          200
      }
      time_to_live  5
    }
  }
}
```

It will look like this in the *ParticleUniverseViewer* application:



Notice that, compared to [tutorial.2.2](#), [tutorial.2.3](#) doesn't contain settings in the technique for default particle width and height anymore. Instead, the emitter contains the attribute 'all_particle_dimensions' that also defines particle width and height (only for particles that are emitted by that particular emitter). The difference is that 'all_particle_dimensions' makes it possible to define a different value for width and height per particle. In the [tutorial.2.3](#) example, a random value between 'min' and 'max' is assigned, so every particle will get a random width and height between 20 and 80.

Something similar is done with 'velocity', so the 'velocity' is not the same for every particle, but varies between 40 and 200. Note, that velocity is set once during creation of the particle, after which the velocity remains constant.

So let's try this (add changes in red):

```
system tutorial.2.3
{
  technique
  {
    ...

    emitter                               Point
    {
      ...

      emission_rate                         dyn_curved_linear
      {
        control_point                       0 2
        control_point                       3 5
        control_point                       10 40
      }
    }
  }
}
```

The emission rate gradually increases over time (since the particle system was started) according to a curve (line) that is set by three control points.

- At the start the emission rate is 2,
- after 3 seconds the emission rate is 5 and
- after 10 seconds the emission rate is 40.

All values in between are linear interpolated.

2.4 Add some movement

The examples in the previous paragraph are very limited and need to be spiced up. To affect movement of the particles there are several particle affectors available. The *GravityAffector*, *LinearForceAffector*, *Randomiser*, *JetAffector* and *VelocityMatchingAffector* are just a few examples of affectors that influence the movement of a particle. Zero or more particle affectors can be added to one technique. If you add multiple affectors to a technique, the total effect is combined and applied to the particles. Lets take a look at one of the previous examples and add some affectors.

```
system tutorial.2.4
{
  technique
  {
    renderer                               Billboard {}
    material                               ParticleUniverse/GreenBall
    default_particle_width                 20
    default_particle_height                20
    visual_particle_quota                  100
  }
}
```

```

emitter                                     Point
{
    emission_rate                           40
    angle                                    60
    direction                                0 1 0
    velocity                                  100
    time_to_live                             3
}

affector                                    LinearForce
{
    force_aff_vector                         0 -100 0
}

affector                                    Randomiser
{
    rand_aff_max_deviation_x                 50
    rand_aff_time_step                       0.2
}
}

```

This is [tutorial.2.2](#) where a *LinearForceAffector* and a *Randomiser* are added. The result is that particles are forced to a direction that is defined as a 3D vector (0, -100, 0). This means that particles are forced to move down the y-axis. On top of that, the particle direction is randomised. Every 0.2 seconds the x-factor of each particle direction is affected with a force that lies between [-50, +50].

Important remark:

Version 0.8 and 0.81 still contain a bug in the *Randomiser*, which is already solved in version 1.0. Change in file *ParticleUniverseRandomiser.cpp* the function `Randomiser::_affect()`.

Change line

```
particle->direction = Ogre::Vector3(Ogre::Math::RangeRandom(-mMaxDeviationX,
mMaxDeviationX), Ogre::Math::RangeRandom(-mMaxDeviationY, mMaxDeviationY),
Ogre::Math::RangeRandom(-mMaxDeviationZ, mMaxDeviationZ));
```

into

```
particle->direction += Ogre::Vector3(Ogre::Math::RangeRandom(-
mMaxDeviationX, mMaxDeviationX), Ogre::Math::RangeRandom(-mMaxDeviationY,
mMaxDeviationY), Ogre::Math::RangeRandom(-mMaxDeviationZ, mMaxDeviationZ));
```

and recompile (otherwise the example gives unexpected results).

2.5 Colour your particles

Applying some colour to your particles can be done in different ways. Take a look at the next example ([tutorial.2.5](#)) and especially the two red lines. This two lines indicate that as soon as a particle is emitted, a colour is applied that lies between 0, 0, 1, 1 and 1, 1, 0, 1. Because the notation of the colour is *red green blue [alpha]*¹ the interval is a generated colour that lies somewhere between 100% blue and 100% yellow (red + green gives yellow).

Applying the colour depends on the material. The material should have the property 'lighting off' otherwise the colour cannot be applied.

¹ Red, green, blue and alpha have a value between [0, 1]

```

system tutorial.2.5
{
  technique
  {
    renderer          Billboard {}
    material          ParticleUniverse/ExplosionFire
    default_particle_width 60
    default_particle_height 60
    visual_particle_quota 100

    emitter          Point
    {
      emission_rate 40
      angle         60
      direction     0 1 0
      velocity      100
      time_to_live  3
      start_colour_range 0 0 1 1
      end_colour_range 1 1 0 1
    }
  }
}

```

Run the *ParticleUniverseViewer* application and start [tutorial.2.5](#).

As you can see, the particles have different colours, but as soon as the particle expires it is destroyed immediately, which gives a restless effect. Wouldn't it be nice to fade the particles gradually?

This is possible with the use of the *ColourAffector*. The *ColourAffector* colours the particle during its lifetime. Comparable with the control points that are used in [tutorial.2.3](#) we can add a time/colour attribute. Add the following *ColourAffector* to the technique of [tutorial.2.5](#).

```

system tutorial.2.5
{
  technique
  {
    ...

    emitter          Point
    {
      ...
    }

    affector        Colour
    {
      colour_aff_time_colour 0 1 1 1
      colour_aff_time_colour 1 0 0 0
      colour_operation      multiply
    }
  }
}

```

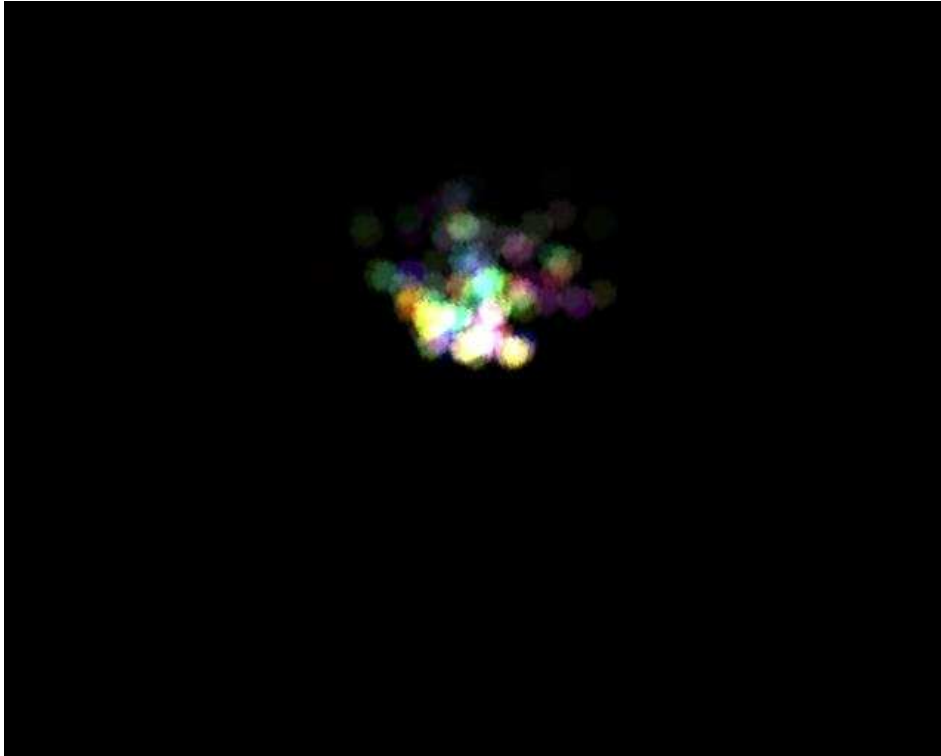
The *ColourAffector* contains two 'colour_aff_time_colour' attributes.

- The first one defines the colour 'white' (100% red + 100% green + 100 % blue) at time '0' of the particle lifetime.
- The second one defines the colour 'black' (0% red + 0% green + 0 % blue) at time '1' of the particle lifetime. Note, that time '1' means the fraction of the particle lifetime.

Because the 'colour_operation' is set to 'multiply' this means that the colour of the particle is multiplied with the colour of the *ColourAffector*. A basic calculation learns that if a particle is red (1, 0, 0) as soon as it is emitted and multiplied with (1, 1, 1), this results in (1, 0, 0); red again. At the end of the particles' lifetime, its colour is multiplied with (0, 0, 0), which results

in (0, 0, 0); black. Concluded, during its lifetime the particle colour turns from red to black and because the 'scene_blend' method of the particle material is 'add' this gives a fading effect.

This results in the picture below:



Change the 'colour_operation' from 'multiply' into 'set' and take a look at the result. Can you explain it?

What is happening with the 'set' colour operation is that it sets the particle colour and therefore overrides the colour that has been set with 'start_colour_range' and 'end_colour_range'. The result is that the particle colour changes from white to black during its lifetime.

2.6 Observing and event handling

Besides emitting particles and affecting them, Particle Universe also has capabilities to observe particles and handle a particular event if it occurs. The observers that are included in the Particle Universe package aren't restricted to observing particles only. The *OnTimeObserver* checks if a certain time period has been exceeded, while the *OnRandomObserver* checks if a generated random value exceeds a predefined threshold. A particle observer must be added to a technique.

Take a look at the example below. The particles are emitted with a velocity between 100 and 200 and also get affected by the *LinearForceAffector*. The *OnVelocityObserver* validates each particle and if the velocity exceeds 200, it will call the (event) handler. In this example the *DoFreezeEventHandler* is called, which immediately stops movement of the particle. Each observer may contain multiple event handlers.

```

system tutorial.2.6
{
  technique
  {
    renderer          Billboard {}
    material          ParticleUniverse/GreenBall
    default_particle_width 40
    default_particle_height 40
    visual_particle_quota 500

    emitter          Point
    {
      emission_rate 30
      angle         360
      direction     0 1 0
      velocity      dyn_random
      {
        min        100
        max        200
      }
      time_to_live 5
    }

    affector          LinearForce
    {
      force_aff_vector 0 -100 0
    }

    observer          OnVelocity
    {
      velocity_threshold greater_than 200
      handler            DoFreeze {}
    }
  }
}

```

Note: While testing the script I found a small bug; if a particle gets freezed and expires, the freeze flag remains true. This means that when it is pulled out of the pool and emitted again, it doesn't move. If the [tutorial.2.6](#) script is started, emission 'stops' after the 500th particle. This is solved in the next release.

2.7 Basic Collision

Particle Universe includes basic particle collision although if you want to do it right you rather should use a collision library. Particle Universe V1.0 has PhysX support, which provides better collision than the build-in one.

2.7.1 Colliders en Externs

A collider is a virtual shape with which a particle can collide. The shape cannot be seen, but it represents an object in the scene that is visible. For example, if the scene contains a building mesh (entity) on position (x, y, z) you can add a *BoxCollider* on this same location and with roughly the same dimensions. If a particle hits the *BoxCollider*, it changes direction.

The Particle Universe package contains three colliders, the *SphereCollider*, the *PlaneCollider* and the *BoxCollider*. These colliders are in fact affectors. They can be added to a technique and only the particles of that technique (or better, the particles that are emitted by emitters that are added to the technique) are affected when they hit the collider.

This can be a bit complicated if your particle system is moving and the collider doesn't or vice versa. That is the reason why the three colliders have a 'Extern' counterpart. An 'Extern'

is a component outside the particle system, but does perform actions on that particle system. The three Extern colliders and the PhysXExtern are a few examples.

In contrast with the *SphereCollider*, *PlaneCollider* and *BoxCollider*, the *SphereColliderExtern*, *PlaneColliderExtern* and *BoxColliderExtern* are *Ogre::MovableObjects* so they can be attached to a particular *Ogre::SceneNode* (for example the *Ogre::SceneNode* to which a building is attached). On the other hand, they can also be added to a technique (as Extern), so the Extern collider (which is attached to one node) still affects the particle system (which is attached to another node).

Because using Extern colliders needs some additional (C++) code, the next example is focussed on a *PlaneCollider*.

```
system tutorial.2.7.1
{
    technique
    {
        renderer                Billboard {}
        material                 ParticleUniverse/GreenBall
        default_particle_width   40
        default_particle_height  40
        visual_particle_quota    100

        emitter                  Box
        {
            position             0 300 0
            angle                 20
            emission_rate         5
            box_em_width          200
            box_em_height         1
            box_em_depth          1
            direction              0 -1 0
            velocity              dyn_random
            {
                min               200
                max               300
            }
        }

        time_to_live             5
    }

    affector                    PlaneCollider
    {
        position                 0 -100 0
        plane_collider_normal    0 1 0
        collision_intersection     point
        collision_type            bounce
        collision_bouncyness      0.5
    }

    affector                    LinearForce
    {
        force_aff_vector         0 -500 0
    }
}
}
```

2.7.2 Inter particle collision

Interparticle collision means that particles collide with each other. If you use a brute-force method and your particle system emits 100 particles this means that the number of inter particle collision validations is $100 * 100$. A smart trick is needed to reduce this amount. Particle Universe uses the concept of a Spatial Hashtable.

Assume that you divide your 3D world into cells. A particle in one cell only has to take its neighbours into account to validate collision. Particles in other cells that are further away will definitely not collide, so they won't have to be checked. This reduces the number of validations. The Spatial Hashtable represents these cells and particles that are added to the table, will be added to a particular cell, depending on its position. The complicated factor is that the distribution of cells must be set in such a way, that one cell doesn't contain all particles and that there aren't a lot of cells with only one particle.

To enable interparticle collision, you have to do two things:

1. Add an *InterParticleCollider* to the technique
2. Change the settings of the Spatial Hashtable in the technique (using attributes such as 'spatial_hashing_cell_dimension'), so the distribution of particles over the cells is correct (not all particles in one cell and not a lot of cells with just one particle). Getting the settings right takes some fiddling.

The first example enables interparticle collision. It uses the default values of the Spatial Hashtable, so only the *InterParticleCollider* is added to the technique.

```
system tutorial.2.7.2
{
  technique
  {
    renderer          Billboard {}
    material          ParticleUniverse/GreenBall
    default_particle_width 40
    default_particle_height 40
    visual_particle_quota 100

    emitter          Box
    {
      emission_rate 10
      box_em_width 200
      box_em_height 1
      box_em_depth 1
      direction      0 1 0
      velocity       dyn_random
      {
        min 10
        max 200
      }
    }

    time_to_live 3

    affector      InterParticleCollider {}
  }
}
```

2.7.3 Collision observer

The *OnCollision observer* observes whether a particle has collided either with a *Box*-, *Sphere*-, or *PlaneCollider* or with another particle (inter-particle collision). It triggers an event handler to perform some action.

The script below may look complex, because it includes two techniques. The first technique covers the falling green balls and defines the *OnCollision observer*. If a particle hits the *PlaneCollider*, it executes the *DoPlacementParticle handler* and the *DoExpire handler*. The latter just expires the particle, but the first handler places the Sparks emitter on the collision position and let it emit 5 'sparkle' particles.

The second technique includes the Sparks emitter and the Billboard renderer that renders the 'sparks'. The Sparks emitter doesn't emit until it is activated by the the *DoPlacementParticle handler*; the emission_rate is set to 0.

```

system tutorial.2.7.3
{
    technique
    {
        renderer                Billboard {}
        material                 ParticleUniverse/GreenBall
        default_particle_width   40
        default_particle_height  40
        visual_particle_quota    100

        emitter                 Box
        {
            position             0 300 0
            angle                 20
            emission_rate        10
            box_em_width         200
            box_em_height        1
            box_em_depth         1
            direction             0 -1 0
            velocity              dyn_random
            {
                min              200
                max              300
            }

            time_to_live         5
        }

        affector                PlaneCollider
        {
            position             0 -100 0
            plane_collider_normal 0 1 0
            collision_intersection point
            collision_type        bounce
        }

        observer                OnCollision
        {
            handler              DoPlacementParticle
            {
                force_emitter    Sparks
                number_of_particles 5
            }
            handler              DoExpire
            {
            }
        }
    }

    technique
    {
        visual_particle_quota    500
        material                 ParticleUniverse/Star
        default_particle_width   20
        default_particle_height  20
        renderer                 Billboard
        {
        }
        emitter                 Point    Sparks
        {
            // Set the emission_rate default to 0 to prevent auto-emission
            emission_rate        0
            angle                 360
            direction             0 -1 0
            velocity              130
            time_to_live         0.4
        }
    }
}

```

3 Advanced options

3.1 Multiple techniques and renderers

Script [tutorial.2.7.3](#) shows a particle system with 2 techniques. A particle technique has its own rendererer, so with the use of multiple techniques in one particle system, it is possible to render particles with different shapes and different materials. This allows for complex particle systems, with all complexity integrated into one container, the particle system.

3.2 Particle System LOD

Particle System LOD (= Level Of Detail) is used to reduce the resource usage of a particle system so performance will not decrease too much. For example. it is not needed to render Entities (meshes) if the distance between the camera and the particle system is large. The users won't see the difference if particles at a large distance are rendered as Billboards instead. Important to realise is that with a multiple camera setup the LOD feature doesn't work. The benefit to support multiple camera LOD seems lesser, because the complexity becomes larger. If the scene has multiple cameras, you also must set the name of the main camera in the particle system (in script this is done with 'main_camera_name').

Particle Universe supports two ways of LOD:

- discrete LOD, making use of multiple techniques
- using a ramp function.

3.2.1 Discrete LOD

This is similar to the way LOD is used in Ogre's material system.

- First, the attribute 'lod_distances' must be set in the particle system. This attribute controls the distances at which different (particle) techniques can come into effect.
- Second, a level-of-detail (LOD) index is assigned to each technique in the particle system. The distance at which a LOD level is applied is determined by the lod_distances attribute of the containing particle system.

The lay-out of a particle system with three discrete LOD-levels looks like this.

```
system tutorial.3.2.1
{
    lod_distances           800 2200
    smooth_lod             true
    main_camera_name       PlayerCam

    technique
    {
        lod_index          0
        ...
    }

    technique
    {
        lod_index          1
        ...
    }

    technique
    {
        lod_index          2
    }
}
```

```

    }
}

```

The 'smooth_lod' attribute is to apply a smoothness of the transition. As soon as the LOD transition of another technique comes into effect, the already emitted particles of the previous technique are still alive and shouldn't be destroyed immediately. The 'smooth_lod' makes sure that these particles still exist even when another technique is activated. These particles stay alive until they finally expire.

The 'main_camera_name' called "PlayerCam" is the one that is used to calculate the distance from the particle system, which is used to execute LOD transitions.

The technique with index 0 is activated if the distance between the particle system and the "PlayerCam" is between 0-800 units.

The technique with index 1 is activated if the distance between the particle system and the "PlayerCam" is between 800-2200 units.

The technique with index 2 is activated if the distance between the particle system and the "PlayerCam" is larger than 2200 units.

3.2.2 Ramp function

Instead of discrete LOD-levels, it is also possible to influence particular attributes if the camera moves. In the example below, the 'emission_rate' increases when the camera becomes nearer and decreases when the camera moves away. This is done by means of the 'camera_dependency' and the insertion operator <<

```

system tutorial.3.2.2
{
    main_camera_name      PlayerCam
    technique
    {
        renderer Billboard {}
        material           ParticleUniverse/GreenBall
        default_particle_width 40
        default_particle_height 40
        emitter           Point
        {
            emission_rate 30 << camera_dependency
            {
                distance_threshold 400
                increase           false
            }
            direction      0 -1 0
            velocity       120
            time_to_live   dyn_random
            {
                min        2
                max        4
            }
        }
    }
}

```

3.3 Emission of non-visual particles

Besides emission of visual particles, it is also possible to emit non-visual particles. Several components in Particle Universe are considered particles and can be emitted. These components are:

- Particle System, identified by type 'system_particle'
- Particle Technique, identified by type 'technique_particle'
- Particle Emitter, identified by type 'emitter_particle'
- Particle Affector, identified by type 'affector_particle'
- Visual Particle, identified by type 'visual_particle'

Emitting particle types other than visual particles is done by using the keyword 'emits' in the emitter definition (emission of visual particles is default, so you can omit the 'emits' keyword). The (script) name of the particle type must also be supplied.

- The name is searched in the same technique if it concerns an 'emitter_particle' or an 'affector_particle'.
- The name is searched in the same particle system if it concerns a 'technique_particle'.
- If the particle type is a 'system_particle', the name is searched in the available particle system templates.

A basic script that emits other emitters looks like this.

```
system tutorial.3.3
{
  technique
  {
    default_particle_width      10
    default_particle_height     10
    emitted_emitter_quota       10
    renderer Billboard {}
    material                     ParticleUniverse/GreenBall

    emitter                     Point
    {
      emits                     emitter_particle Balls
      emission_rate             2
      angle                     360
      direction                 0 1 0
      velocity                  300
      time_to_live              3
    }

    emitter                     Point Balls
    {
      emission_rate             30
      time_to_live              0.5
    }
  }
}
```

3.4 Atlas texture and TextureAnimator (V1.0 only)

Atlas textures are supported in version 1.0. Their use is very convenient, because it gives the possibility to assign a different image to each particle individually. In general, the texture of a particle is always identical for particles in a technique. This is, because only one material can be used in a technique. However, if you use different uv-sets (the coordinates of a texture) for each particle, the particles appear different.

Defining these uv-coordinates is done in the renderer by using the keywords 'texture_coords_rows' and 'texture_coords_columns'. They divide the texture in a matrix of coordinate sets. Each set gets its own index. So, if 'texture_coords_rows' is set to 4 and 'texture_coords_columns' is set to 4, the texture is divided in a 4x4 matrix with coordinate sets [0..15].

With the use of 'start_texture_coords' and 'end_texture_coords' in the emitter, it is possible to randomly assign a texture coordinate set (uv-set) to a particle. A small example:

```
system tutorial.3.4
{
  technique
  {
    default_particle_width      40
    default_particle_height     40
    emitted_emitter_quota      10
    material                    ParticleUniverse/Interpolate
    renderer Billboard
    {
      texture_coords_rows      6
      texture_coords_columns   6
    }

    emitter Point
    {
      emission_rate            10
      angle                    360
      direction                 0 1 0
      velocity                  200
      time_to_live              3
      start_texture_coords      0
      end_texture_coords        15
    }
  }
}
```

You can take this one step further and add a *TextureAnimator*. The *TextureAnimator* is an affector that displays different uv-sets on a particle as if it was playing a movie. If a *TextureAnimator* was added to the script above and it was set to loop through uv-sets [0..16] you saw an animation on each particle.

The Particle Universe package contains a script (example_029), that does just that. To easily create atlas textures, there is an *AtlasTextureTool* added to the Particle Universe package, which stitches a number of images together and creates one big image containing all the images.

3.5 Using PhysX (V1.0 only)

Particle Universe includes PhysX support from version 1.0. This means, that particles behave as real physical objects. In the first release this concerns the primitive objects Sphere, Box and Capsule. PhysX is responsible for the physical behaviour, while Particle Universe is mainly used as a render system and makes it possible to create particle scripts with PhysX attributes. Of course the integration of both libraries is transparent and creation of a PhysX enabled particle system is as easy as creating one without PhysX support. Although there are only a few physical primitives supported, it doesn't mean that there is a restriction in the rendered shape. You can render real meshes (entities) while their physical shape is a Capsule. Setting up Particle Universe to support the PhysX library is covered in detail in the manual, so it isn't repeated here.

If you want to use PhysX you have to code a few lines; not everything can be done by means of a script:

1. **Setup (initialise) PhysX:** The assumption is, that setting up PhysX is done outside of the Particle Universe plugin, but there is an `initNx()` and `exitNx()` function for test purposes though. Initialising PhysX is done in your client application.
2. **Set the NxScene:** When the NxScene is created in your client application, the function `ParticleUniverse::PhysXBridge::setScene(NxScene* scene)` must be called, otherwise it isn't possible to create PhysX actors or fluid. The PhysXBridge is a singleton class that interfaces between your PhysX driven client application and Particle Universe.
3. **Define collision between groups:** Particles must be assigned to a group to enable (inter-particle) collision with Nx-actors from the same group. For example
`nxScene->setActorGroupPairFlags(100, 100, NX_NOTIFY_ON_START_TOUCH);` If your particles are assigned to group 100, it means that they collide with each other.
4. **Synchronize Particle Universe with PhysX:** This must be done in your client application every x milliseconds (= `timeElapsed`). Synchronizing is done by means of the function `ParticleUniverse::PhysXBridge::synchronize(Ogre::Real timeElapsed)`

That's it. More code isn't needed and makes it possible to execute a PhysX enabled script like the example below:

```

system tutorial.3.5
{
    technique
    {
        visual_particle_quota          1000
        position                       0 1000 0
        material                       ParticleUniverse/Crate
        renderer                       Box {}

        emitter                        Box
        {
            box_em_width                400
            box_em_height               400
            box_em_depth                400
            all_particle_dimensions     dyn_random
            {
                min                    10
                max                    80
            }
            emission_rate                10
            time_to_live                 10
            velocity                     100
            angle                        40
            direction                    0 -1 0
            mass                         100
            range_start_orientation     0.2 1 2 3
            range_end_orientation       3 3 2 1
        }

        extern                        PhysX
        {
            physx_actor_group           100
            physx_shape                 Box
            {
                physx_angular_velocity  0 10 0
                physx_angular_damping   0.5
            }
        }
    }
}

```

This script emits falling box-shaped particles.

If you also want to use the *OnCollision observer*, some additional coding has to be done. Particle Universe doesn't automatically detect that PhysX particles are colliding. Usually a contact reporter is created to detect collision. Particle Universe has to option to use its build in

contact reporter for testing, but in most cases a contact reporter is created by the client application. Assume this is the case, a call from the clients' contact reporter to the function

```
ParticleUniverse::PhysXBridge::onContactNotify(NxContactPair& pair, NxU32 events, NxVec3 contactPoint)
```

must be made.

3.5.1 Fluid

Fluid is also supported, but the fluid attributes cannot be set by means of a script. This is not a technical limitation; fluid isn't used that much in realtime applications. If fluid becomes more mainstream, Particle Universe will enable the use of fluid attributes in its scripts.

For the time being, some additional coding is needed. The easiest way is to create a script with an empty *PhysXExtern*.

```
system tutorial.3.5.1
{
    technique
    {
        visual_particle_quota          1000
        default_particle_width         40
        default_particle_height        40
        position                        0 800 0
        material                        ParticleUniverse/GreenBall
        renderer                        Billboard {sorting true}

        emitter                         Point
        {
            emission_rate              40
            time_to_live                100
            velocity                    500
            angle                       10
            direction                   1 -1 0
            mass                         1
        }

        extern                          PhysX
        {
            // Leave it empty and create a fluid by code, using this extern
        }
    }
}
```

If the application starts and PhysX is initialised, the creation of a fluid is done by means of code. An example:

```
/** Create the particle system.
 */
ParticleUniverse::ParticleSystem mySys = ParticleUniverse::ParticleSystemManager::
getSingleton().createParticleSystem("myPhysXSystem", "tutorial.3.5.1", sceneManager);

...

/** Call prepare() before createFluid(), because communication with the PhysX library
    is setup in the prepare().
 */
mySys->prepare();
PhysXExtern* physXExtern = static_cast<PhysXExtern*>(sys->getTechnique(0)->getExtern(0));
NxFluidDesc nxFluidDesc; // PhysX class
nxFluidDesc.kernelRadiusMultiplier = 2.3f;
nxFluidDesc.restParticlesPerMeter = 3.0f;
nxFluidDesc.stiffness = 200.0f;
nxFluidDesc.viscosity = 300.0f;
nxFluidDesc.simulationMethod = NX_F_SPH;
```

```
physxExtern->createFluid(nxFluidDesc);  
mySys->start();
```

4 Complex particle systems

4.1 Flocking behaviour

Particle Universe includes some effectors, which in combination, perform flocking behaviour. Flocking is “*the collective motion of a large number of self-propelled entities*”, exhibited by birds, insects and fish.

Flocking behaviour consists of three components:

- Separation - avoid crowding neighbours (short range repulsion)
- Alignment - steer towards average heading of neighbours
- Cohesion - steer towards average position of neighbours (long range attraction)

These three components are implemented in Particle Universe by means of the

- *CollisionAvoidance effector* (separation)
- *VelocityMatching effector* (alignment)
- *FlockCentering effector* (cohesion)

example_027 in the Particle Universe plugin package shows how those three effectors are used in combination. Both the *CollisionAvoidance effector* and the *VelocityMatching effector* make use of the Spatial Hashtable functionality, so the Spatial Hashing properties have to be set correct in the technique for optimal performance.

4.2 Flip Flop

In some situations a mechanism is needed where multiple forces, that change over time, are applied to a particle system. Generation of wind is such an example. This can be done by using some kind of perbutation / noise effector, but because Particle Universe omits this, a flip flop mechanism can be used instead.

The basic idea is that multiple effectors are enabled or disabled in case of a certain event. Take a look at the following excerpt.

```
system tutorial.4.2
{
  ...
  technique
  {
    ...
    observer OnRandom
    {
      observe_interval 1
      random_threshold 0.5
      handler DoEnableComponent
      {
        enable_component affector_component WindLeft true
      }
      handler DoEnableComponent
      {
        enable_component affector_component WindRight false
      }
    }
    observer OnRandom
    {
```

```

        observe_interval          1
        random_threshold          0.6
        handler                   DoEnableComponent
        {
            enable_component      affector_component  WindRight  true
        }
        handler                   DoEnableComponent
        {
            enable_component      affector_component  WindLeft   false
        }
    }

    affector                      LinearForce  WindLeft
    {
        enabled                   false
        force_aff_vector          -150 -50 0
    }
    affector                      LinearForce  WindRight
    {
        enabled                   false
        force_aff_vector          50 0 0
    }
}

```

The two *OnRandom observers* are generating a random value and as soon as the value exceeds a threshold, the event handlers are activated. The trick is, that the event handlers included in the observers are doing an opposite action. The first one enables the *WindLeft* affector and disables the *WindRight* affector (both *LinearForce* *affectors*), the second observer does it the other way around. This creates *Wind-like* forces.

4.3 Explosion

Particle systems are associated with explosions, so this tutorial is not complete if explosions aren't covered. An explosion comes in all types and sizes, but it doesn't matter which type you want to create, it is very difficult to produce a convincing explosion.

One type of explosion has in common that it starts with a flash of bright light, that expands quickly, while hot debris is shooting in every direction. The debris also has an initial high velocity, but quickly loses speed. The bright light fades and is slowly replaced by smoke. So we need a bright flash, debris and smoke. This requires three materials, so the particle system will also contain three techniques.

The script below is a particle script that defines the flash.

```

system tutorial.4.3
{
    technique                      Fire
    {
        material                   ParticleUniverse/Nucleus
        visual_particle_quota      300
        renderer Billboard
        {
            render_queue_group     50
        }
    }

    emitter                       Point FireEmitter
    {
        emission_rate              60
        angle                      360
        direction                  0 1 0
        velocity                   100
        time_to_live               1
        all_particle_dimensions    dyn_random
    }
}

```

```

        min                20
        max                80
    }
    colour                 1.0 0.5 0.3
    duration               0.3
}

affector                  Scale
{
    xyz_scale              dyn_curved_linear
    {
        control_point     0 2000
        control_point     0.3 20
        control_point     1 0
    }
}

affector                  Colour
{
    colour_operation       multiply
    colour_aff_time_colour 0 1 1 1
    colour_aff_time_colour 0.05 0.3 0.3 0.3
    colour_aff_time_colour 1 0 0 0
}
}
}

```

It uses an emitter that emits only 0.3 seconds (defined by 'duration') and emits 60 particles per second in every direction. The x and y dimensions of the flash particles are between 20 and 80 units, but because of the inclusion of a *Scale affector* they increase rapidly. The *Scale affector* expands the particle size initially at a scale velocity of 2000, but after 0.3 seconds the particle size only increases with a velocity of 20, and comes to a complete halt after 1 second.

The *Colour affector* causes the particle to fade its colour quickly.

Run the script [tutorial.4.3](#) and see what happens. Now add some debris. Therefore the script is expanded with a second technique (in red).

```

system tutorial.4.3
{
    technique              Fire
    {
        material           ParticleUniverse/Nucleus
        visual_particle_quota 300
        renderer Billboard
        {
            render_queue_group 50
        }
    }

    emitter                Point FireEmitter
    {
        emission_rate      60
        angle               360
        direction           0 1 0
        velocity            100
        time_to_live        1
        all_particle_dimensions dyn_random
        {
            min             20
            max             80
        }
        colour              1.0 0.5 0.3
        duration            0.3
    }

    affector                Scale
    {
        xyz_scale           dyn_curved_linear
    }
}

```

```

        control_point      0 2000
        control_point      0.3 20
        control_point      1 0
    }
}

affector      Colour
{
    colour_operation      multiply
    colour_aff_time_colour      0 1 1 1
    colour_aff_time_colour      0.05 0.3 0.3 0.3
    colour_aff_time_colour      1 0 0 0
}

technique
{
    material      ParticleUniverse/Debris
    visual_particle_quota      100
    renderer      Billboard
    {
        billboard_type      oriented_self
        render_queue_group      50
    }

    emitter      Point Debris
    {
        enabled      false
        emission_rate      100
        force_emission      true
        direction      0 1 0
        angle      360
        time_to_live      2
        particle_height      dyn_random
        {
            min      50
            max      100
        }
        particle_width      dyn_random
        {
            min      3
            max      10
        }
        velocity      dyn_random
        {
            min      600
            max      1200
        }
    }

    affector      Scale
    {
        y_scale      dyn_curved_linear
        {
            control_point      0 -200
            control_point      0.3 -20
            control_point      1 0
        }
    }

    affector      Jet
    {
        jet_aff_accel      -2
    }

    affector      LinearForce
    {
        force_aff_vector      0 -100 0
    }

    affector      Colour
    {
        colour_aff_time_colour      0 0.8 1 1
        colour_aff_time_colour      1 0 0 0
    }
}

```

```

observer
{
    on_time
    since_start_system
    handler
    {
        enable_component
    }
}

OnTime
greater_than 0.1
true
DoEnableComponent
emitter_component Debris true
}
}

```

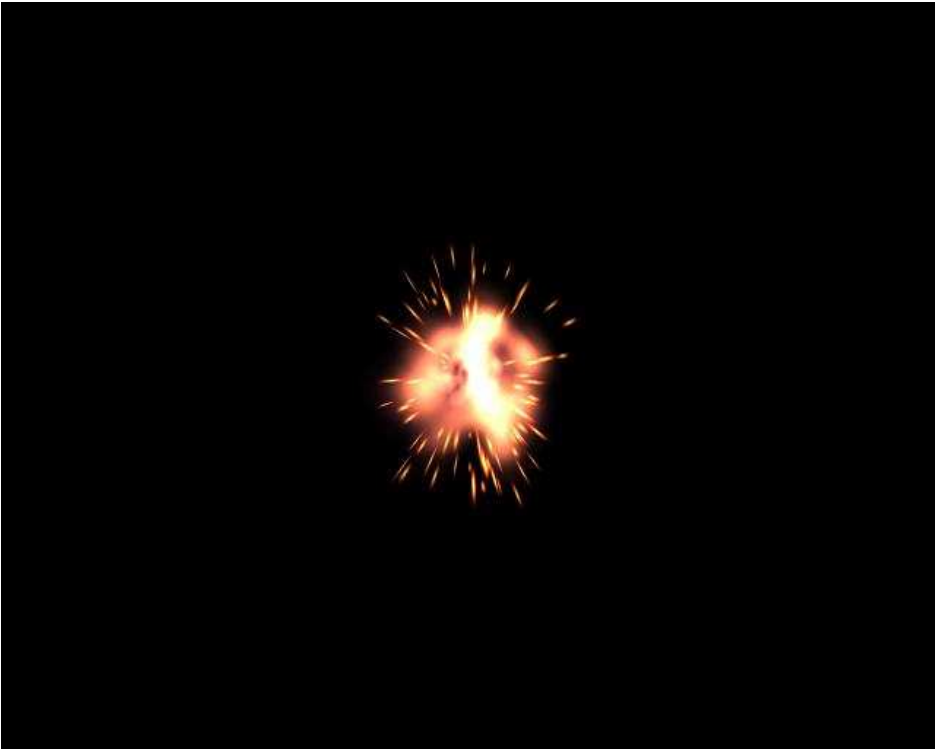
If you look at the debris part, you notice an emitter that has set 'force_emission' to true. This means that the 100 particles (= value of the 'emission_rate') are emitted at once and then no more particles are emitted. The particles are rendered as a Billboard and because 'billboard_type' is set to 'oriented_self' it means that the particles are oriented according to their direction.

The *Scale affector* is also used in this technique, but instead of increasing the size of the particles, they are decreased. This makes sense, because the glowing debris particles cool off during their flight so their glowing trail also decreases.

The *Jet affector* is used to slow the velocity of the debris. The initial velocity of the particles is high, but because of the negative acceleration factor of the *Jet affector*, the particle velocity is decreased rapidly.

Another point of attention goes to the *OnTime observer*. This one enables the Debris emitter as soon as 0.1 seconds have passed since the start of the particle system. It looks better when there is a small delay in the emission of the debris.

Run script [tutorial.4.3](#) again and you should see something like the image below.



The last addition is 'smoke'. If the explosion ends, smoke appears.
 Another technique is added (in red).

```

system tutorial.4.3
{
  technique
  {
    material
    visual_particle_quota
    renderer Billboard
    {
      render_queue_group
    }
    emitter
    {
      emission_rate
      angle
      direction
      velocity
      time_to_live
      all_particle_dimensions
      {
        min
        max
      }
      colour
      duration
    }
    affector
    {
      xyz_scale
      {
        control_point
        control_point
        control_point
      }
    }
    affector
    {
      colour_operation
      colour_aff_time_colour
      colour_aff_time_colour
      colour_aff_time_colour
    }
  }
  technique
  {
    material
    visual_particle_quota
    renderer
    {
      billboard_type
      render_queue_group
    }
    emitter
    {
      enabled
      emission_rate
      force_emission
      direction
      angle
      time_to_live
      particle_height
      {
        min
        max
      }
      particle_width
    }
  }
}
  
```

```

        min          3
        max          10
    }
    velocity         dyn_random
    {
        min          600
        max          1200
    }
}

affector           Scale
{
    y_scale         dyn_curved_linear
    {
        control_point 0 -200
        control_point 0.3 -20
        control_point 1 0
    }
}

affector           Jet
{
    jet_aff_accel   -2
}

affector           LinearForce
{
    force_aff_vector 0 -100 0
}

affector           Colour
{
    colour_aff_time_colour 0 0.8 1 1
    colour_aff_time_colour 1 0 0 0
}

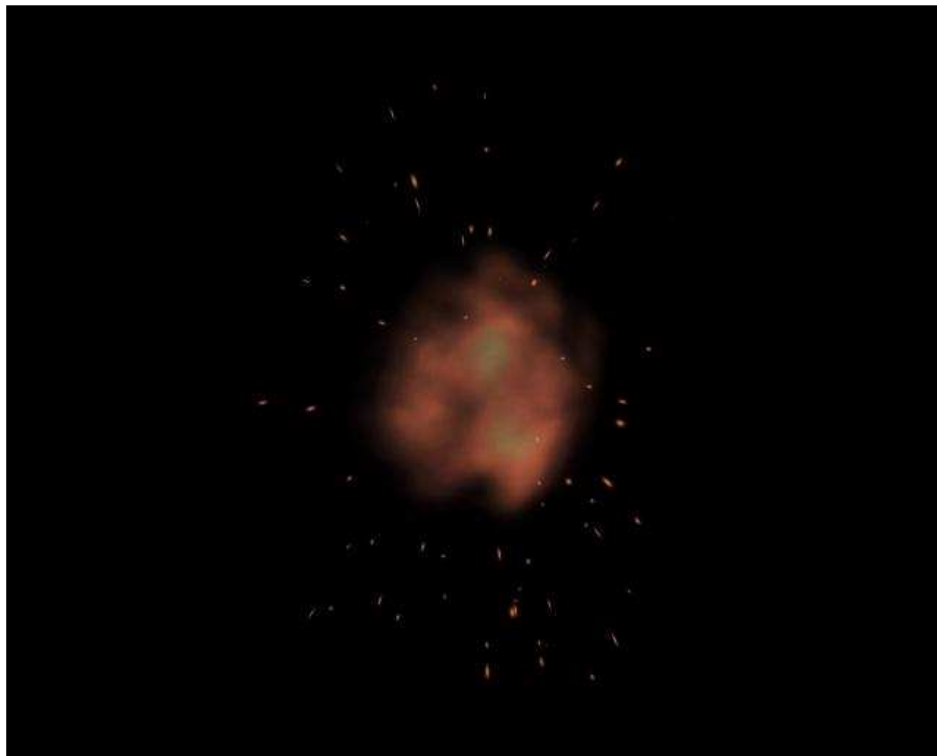
observer           OnTime
{
    on_time         greater_than 0.1
    since_start_system true
    handler         DoEnableComponent
    {
        enable_component emitter_component Debris true
    }
}
}

technique
{
    material         ParticleUniverse/Smoke
    visual_particle_quota 300
    renderer Billboard
    {
        render_queue_group 51
    }
    emitter          Slave
    {
        time_to_live 3
        master_technique_name Fire
        master_emitter_name FireEmitter
        all_particle_dimensions dyn_random
        {
            min 20
            max 80
        }
    }
    affector         Scale
    {
        xyz_scale    dyn_curved_linear
        {
            control_point 0 1200
            control_point 0.2 20
            control_point 1 0
        }
    }
}

```

```
    }  
  }  
  affector          Colour  
  {  
    colour_aff_time_colour    0 0 0 0 0  
    colour_aff_time_colour    0.1 0 0 0 0  
    colour_aff_time_colour    0.11 0.3 0.3 0.3 0.1  
    colour_aff_time_colour    0.5 0.3 0.3 0.3 0.6  
    colour_aff_time_colour    0.8 0.2 0.2 0.2 0.3  
    colour_aff_time_colour    1 0 0 0 0  
  }  
  affector          LinearForce  
  {  
    force_aff_vector          0 25 0  
  }  
}
```

The smoke emitter is a Slave emitter. This means that it emits smoke particles as soon as the emitter with the name 'FireEmitter' of the technique with the name 'Fire' has emitted a particle. The smoke emitter emits a particle on the same position. Adding the smoke emitter results in a particle system that looks like the image below.



To be honest, the quality of the explosion is still questionable. In my opinion the best results are achieved with just a few particles and better artwork. A combination of the *TextureAnimator* and a skilled artist makes the difference.