Particle Systems

Luca Buratti and Andrea Piscitello

November 13, 2014

1 Introduction

The modeling of fuzzy phenomena like clouds, water or smoke can be really difficult for many reasons. First of all these elements does not have a regular shape. They have instead irregular, complex and ill defined surfaces. Furthermore the shape of these object changes a lot during their life, often according to very complex laws that can result very difficult to model properly. For these reasons approaches like polygons rendering with non-procedural motion, can result very difficult to use or even unsuitable. Opposed to conventional methods, particle systems represent a powerful paradigm to model highly chaotic phenomena in a quite simple way. First of all, the particles make these systems highly flexible to model different kind of objects with different consistencies or densities. Moreover the behavior of these system is not defined *a priori* using a simulation-like approach, but is computed procedurally while executing the program. Finally, the intrinsic nature of particle systems seems to perfectly fit the parallel architecture of the GPU allowing to compute the status of many particles at the same time.

2 History

Even if it is possible to find some early examples of particle systems in some video-games of the 60's, the genesis of particles systems has been associated to the paper "Particle Systems: A Technique for Modeling a Class of Fuzzy Objects" [1], published by William T. Reeves in 1983. In this paper Reeves created a new paradigm that enabled the modeling of effects such as fire, snow, rain and clouds. This approach simulates a virtual collection of particles by applying Newton's basic laws of motion, creating computer graphics elements that exhibited fuzzy properties. This new approach differed from the conventional ones in computer graphics in two important ways. First, an object is defined as set of simple particles that define its shape, while usually an object is represented by a set of primitive surface elements such as polygons. Second a particle system is a dynamic entity where its particles change and move with the passage of time. Using this paradigm, Reeves was able to create a wall of

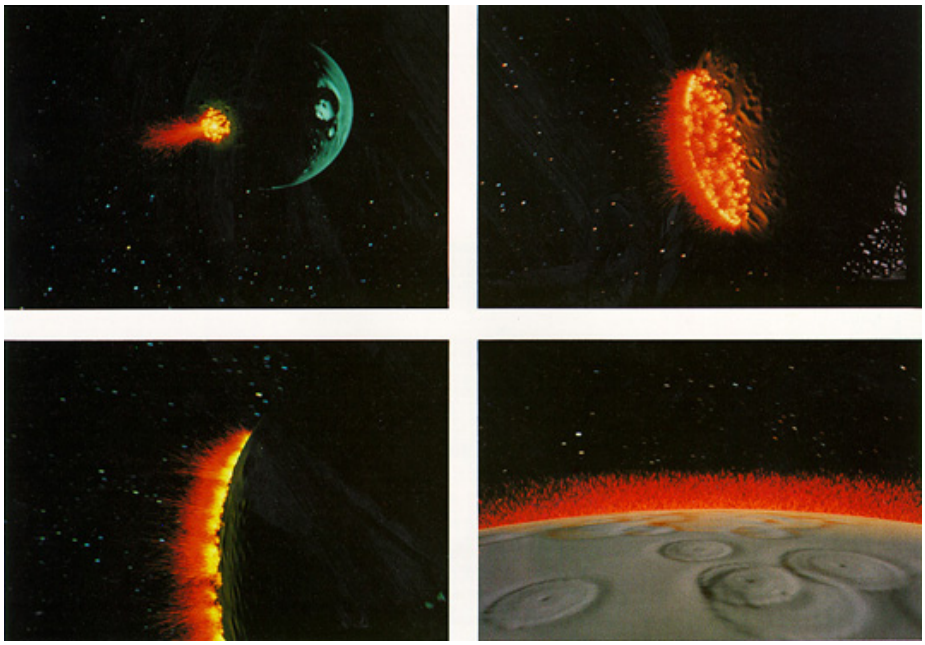

Figure 1: Pictures of the fire of wall scene from *"Star Trek II: The Wrath of Khan"*

fire effect for the film *"Star Trek II: The Wrath of Khan"* and coined the term particle system for this particular effect.

In the past, mostly because of the high computational power requested by particle systems, their exploitation have been constrained to non real-time applications like, for instance, movies. In this field pre-rendering techniques can be applied in order to overpass the power limitations.

In 2004, the implementations of the time were able to handle up to 10,000 particles in real-time simulations and were mostly limited by the transfer of particle data from the main processor to the graphics hardware (GPU) for rendering. Lutz Latta showed that it was possible to handle up to 1 million of particles exploiting a full GPU implementation of a particle system [3].

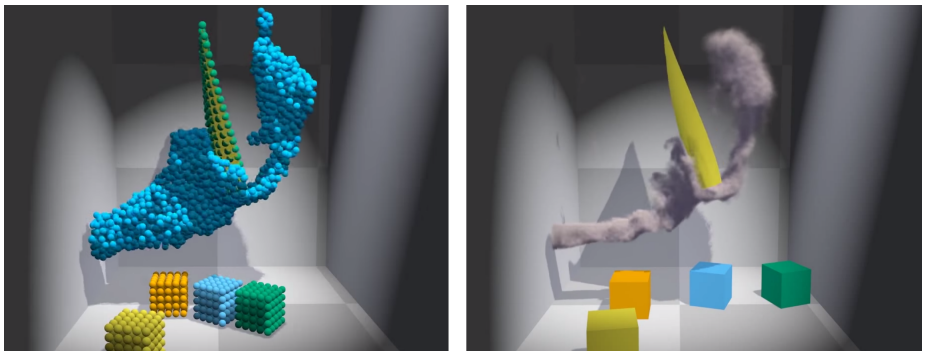

Figure 2: Particle systems with different number of particles. Nvidia FLEX demo, SIGGRAPH 2014

3 Animation of particles

A particle system is composed of many tiny elements that together model a fuzzy object. It creates a volume of particles with individual properties, rather than a surface with a texture. In this way it is possible to change each particle appearance and behavior over time and, by exploiting physic laws, it is possible to give realism to the whole system. The intrinsic nature of particles, easily allow to augment the power of the entire system through physics, rather than using "manual" techniques like, for instance, key-frames. However, modeling a system using only accelerations and forces may result very difficult, so the goal of a particle system is to provide a realistic visualization of a phenomenon, looking for a trade-off between a kinematic control and a physically-based simulation.

In order to let the system evolve properly some parameters are needed. These parameters are often set by means of stochastic processes and for each parameter a range for the particle's value is defined. This range is usually defined by using its mean value and maximum variance.

3.1 Particle Generation

One of the first parameters that are usually chosen to model the system is the number of particles that compose it. This is a very important attribute of the system, since it allows to define the global density of the system and thus it can be used to model dynamic behaviors for different type of materials (water, smoke, fire...). This number can be defined at each frame f with an equation similar to the following:

$$NewParticles_f = MeanParticles_f + Rand() * VarianceParticles_f$$

where *MeanParticles* and *VarianceParticles* have been previously defined and are respectively the mean and the variance of the distribution of the number of

particles to be generated; *Rand()* is a procedure returning a number between -1.0 and 1.0.

The same approach can be used to define attributes of the single particles. The most used attributes are the following:

- Initial position
- Initial velocity (speed and direction)
- Initial size
- Initial color
- Initial shape
- Lifetime

For each of the previous attributes corresponding value can be determined by:

$$InitialValue = MeanValue + Rand() * VarianceValue$$

It is important to highlight that this simple formula can be applied to practically every attribute of the system.

3.2 Particle Evolution

Basic equations for modeling motion in 3d space are provided below (with Position P, Velocity V and Acceleration A):

$$V = V_0 + \int A \mathrm{d}x$$

$$P = P_0 + \int V \mathrm{d}x$$

When updating the state of a particle, for a small time interval Δt these equations can be approximated in order to more easily compute the results:

$$V' = V + A\Delta t$$

$$P' = P + \frac{V+V'}{2}\Delta t$$

Animation operations on particles can either initialize or alter the position or velocity of particles. In a purely physical simulation, these values would first be initialized, then, for each time interval Δt, the velocity would be altered by external accelerations such as gravity, and finally the position would be updated as shown above. However, for kinematically controlled motion, the position may be set directly, regardless of the previous position or velocity. It is also

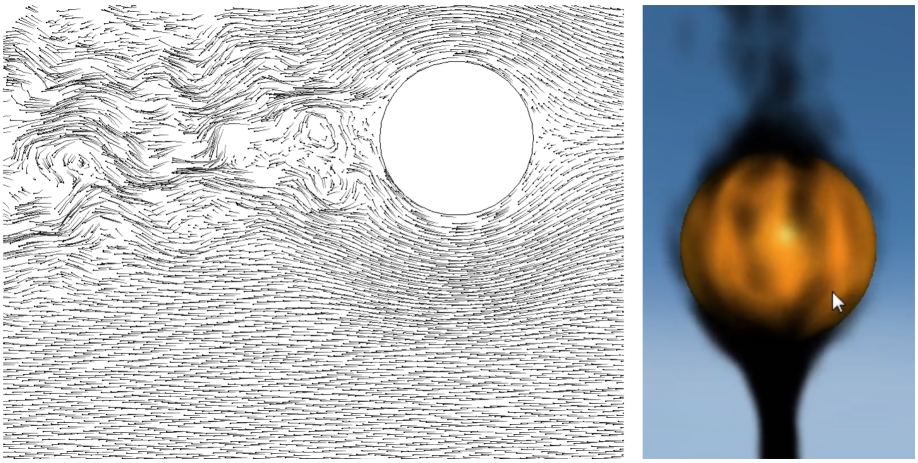

Figure 3: Collision detection used to model smoke on a sphere surface. Source: Curl-Noise for Procedural Fluid Flow [7].

sometimes useful to set the position relative to the previous position or to alter the velocity in ways other than applying a simple translational acceleration.

Other attributes which are less strictly related to the physic behavior of the system like size and color, for instance, can be modeled in a "controlled" way, just defining an empirical law which defines their evolution.

The behavior of a particle can also depend on the one of the near particles. In fact many techniques can be exploited for both motion and appearance attributes. For instance, concerning motion, collision detection algorithms can be implemented in order to produce bumps or crashes. As regards appearance, instead, the light emitted by some particular particle can lighten other ones or some particles can project shadows on other ones.

3.3 Particle Death

A key attribute, that is set at the generation stage, is the lifetime. At every frame after the generation of a particle, its lifetime is decremented and when it reaches zero the particle is killed. Other reasons can produce the death of a particle. For instance, a particle that goes too far, exiting from field of view, is killed as well as a particle which color intensity or transparency goes lower than a determined threshold.

3.4 Particle Rendering

Once everything concerning the existence, behavior and appearance of a particle has been defined, all these information can be exploited to actually represent the particle on the screen. In this phase the attributes that are taken into account are:

- Position (x,y,z)
- Size
- Color (r,g,b)
- Opacity

Additional information can be used in order to exploit, for instance, motion blur techniques. Traditionally, in computer animation, frames have been represented as still images, like instants of time. This approach can be very dangerous and can cause problems like aliasing and strobing effects. For this reason, motion blurring techniques have been introduced to both face this issues and to produce sequences that can result correct to the viewer.

3.5 Particle Hierarchy

An interesting technique, proposed by Reeves [1], is the usage of hierarchical particle systems. This technique consists in the definition of nested systems. At first a particle system is generated, as usually, by modeling it accordingly to the individual attributes of each particle. A single particle of the main system (which will be called super-particle from now on) is in turn a particle system whose particles are initialized with respect to the behavior of the "super-particle". Its behavior is affected by its particles and it results in slightly deviation from the standard behavior of the "super-particle".

4 Some typical uses

In this section some possible particle system uses are highlighted, with a brief logical description of their usual implementation. For some of them we've suggested a particular lecture or article that we've found very interesting.

4.1 Fire

Fire is one of the most complex elements to render in a realistic way. Particle systems are very useful to make it a bit simpler even if some tricks have to be adopted. In the paper "Particle Animation and Rendering Using Data Parallel Computation" by Karl Sims [2] few strategies are described. First of all the general appearance behavior is defined: in the surface of the flame base, new particles are generated with an initial speed direction upward directed. During the life of the particles, they change their color (in a gradient scale from yellow to red) in such a way to simulate the effect of real particles cooling while leaving the source of the fire. The strategy adopted for the trajectory of each particle is to follow a spiral flight whose axis is normal to the surface of the base but slightly perturbed. Particles are grouped together in flickers for realism purposes. The spiral axis slowly rotates to prevent duplicate motion. Also the frequency of particle generation varies in a pseudo-random way in order to better simulate the behavior of a natural fire.

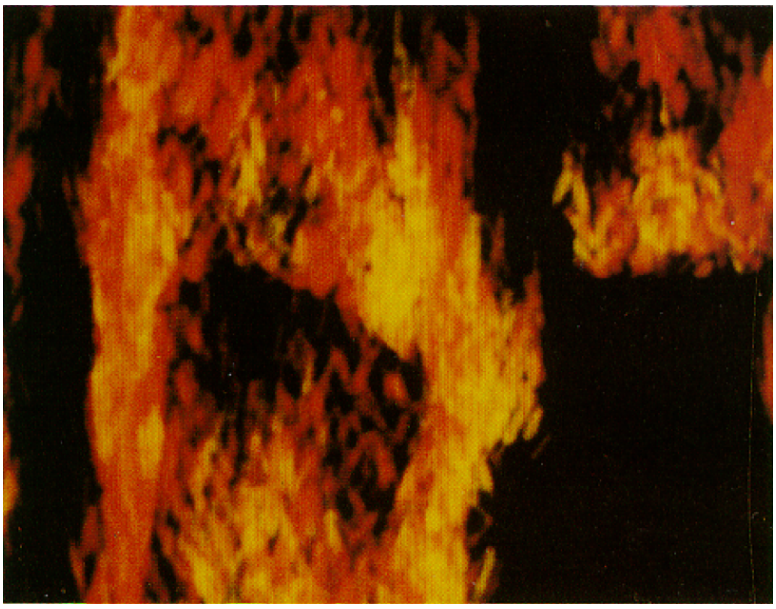

Figure 4: Fire from the paper "Particle Animation and Rendering Using Data Parallel Computation" by Sims [2]

4.2 Snow

Usually snow is modeled using snowflakes particles. These are created and dropped above the scene at each iteration with an initial speed and a main direction straight down with some little random variation. Complex implementation can consider gravity, air friction and the presence of wind in order to model a very realistic behavior. A notable example is the one proposed in the paper "Rendering Falling Rain and Snow" [5] that has been used in the videogame Microsoft Flight Simulator 2004: A Century of Flight. In this approach the use of a double cone for mapping the texture of a particle and the high performance of the system are very interesting.

4.3 Smoke

A simple example of smoke with a particle system is the one proposed in the paper "Smoke Simulation Based on Particle System in Virtual Environments" [6]. We find this implementation interesting and even if it is not very realistic it shows how simple is to achieve good results with not complex particle systems that consider a very limited number of dynamics and effects.

Figure 5: Grass from the paper "Particle Systems - A Technique for Modeling a Class of Fuzzy Object" by Reeves [1]

4.4 Fireworks

There are many possible implementation for fireworks that space from very simple solutions that consider a very limited number of dynamics and effects to very complex systems that model a very wide range of effects. A very interesting solution for this problem is proposed in the Teng-See Loke paper: "Rendering Fireworks Displays" [4]. In this paper the fireworks are modeled considering particular factors such as blinking, Star effect, spinning, mousing and many more. We strongly suggest to read this article if interested in this type of effect.

4.5 Grass

To model grass usually instead of drawing particles as little streaks, the trajectory of each particle over its entire lifetime is drawn. Thus, the time-domain motion of the particle is used to make a static shape. Grasslike green and dark green colors are assigned to the particles which are shaded on the basis of the scene's light sources. Each particle becomes a simple representation of a blade of grass and the particle system as a whole becomes a clump of grass.

5 Our implementation

Studying particle systems, we decided to try to implement a simple one in order to better comprehend how this technique works. We are conscious that the level of our final product is very far from many other examples that can be found on the web, but implementing this simple example helped us to understand how particle systems works and that are very useful and relatively easy to use.

5.1 Overview

We decided to implement a simple flame as a particle system. The system in mainly composed by:

- A single emitter that create 10.000 particles per second, anyway avoiding to create too many particles per frame if there is a very long frame.
- The particles of the system, after being generated, evolve their behavior all along their life. Each particle has 64 different stage of life each one with a different appearance. In order to render the aspect of the particles we used a billboard approach, changing the aspect according to the state of the particle.

The system evolves in two different stages. The first stage, performed by the CPU, generates new particles based on the state of the emitter and updates the existing ones (change their position/direction). If necessary it deletes the particles that have expired. In the second stage the actual rendering of a single particle based on the current life remaining takes place.

5.2 Application stage

This stage can be also divided in three phases, the first one is the tuning of the parameters of a particle when it is created, the second is the update of the particle and the last one is the deletion of all the objects that terminate their life time.

5.2.1 Parameter tuning

Following there is the list of the parameters used for this implementation for modeling the behavior of a single particle and the detailed information about their initialization.

- position: three dimensional vector that specify the current position of a particle, when the object is created this value is initialized according to the emitter.
- speed: three dimensional vector that identify the current speed and direction of a particle. Initialized at a main direction (upward) but perturbed by a small noise value in order to have each particle of the flame that goes

mainly upward, but all the particles together create the characteristic cone at the base of the flame.

- size: float variable that specify the square size of the particle, the particle is rendered as a square with a texture on it. Initialized at a standard dimension perturbed by a little noise.
- life: float variable that identifies the remaining life of a item. Initialized at 5 seconds.
- lifeStage: integer variable that specifies the current life stage of a particle used to render with the correct aspect the object. Value between 0 and 63. Initialized at 0.

5.2.2 Update Model

Each existing particle is updated during this phase, listed below is explained how each parameter of the particle is modified.

- position: Updated according to the speed of the object during the duration of the stage

```
p.pos += p.speed * stageDuration;
```

- speed: Updated when the particles reaches the half of its life.

```
bool change = false;
[...]
if(p.life < 0.5f) {
    if(change == false) {
        p.speed += vec3(-2*p.speed.x, 0, -2*p.speed.z) *
            stageDuration *0.5f;
        change = true;
    }
}
```

The velocity direction is reverted by subtracting twice the values of the x and z components. The *change* variable is used to make this inversion happen just once at the half of the life of a particle.

- size: Unchanged, initially we tried to change the square dimension (so the dimension of the texture rendered) according to the particle remaining life, anyway we didn't liked this approach because the resulting effect was either too many drastic or imperceptible, so we decided to keep a fixed dimension after a particle is generated.
- life: the remaining life is updated as:

```
life = life - (currentTime - lastTime);
```

Where currentTime identify the time stamp of the current frame and lastTime the time stamp of the last frame.

- lifeStage: Updated according the life of the particle in this way:

```
float stageDuration = particleInitialLife/64;
lifeStage = floor((particleInitialLife - life) / stageDuration);
```

Where particleInitialLife represent the maximum life of the particle, stageDuration is the duration of each stage in seconds. So the value of life stage is between 0 and 63.

5.2.3 Deletion of expired particles

In this last simple simulation stage after the life of a particle is updated if it has a value less or equal to 0 it is removed from the scene.

5.3 Shader stage

In this stage the actual aspect of a single particle is computed. The vertex shader implemented, mainly change the orientation of each square using the billboard technique in order to have the square facing exactly the camera. While in the fragment shader the appearance of the particle is chosen based on the state of the particle itself. In order to do so a single "texture Map", with 64 different sub images of a flame during its life, has been used. The sub texture to use is

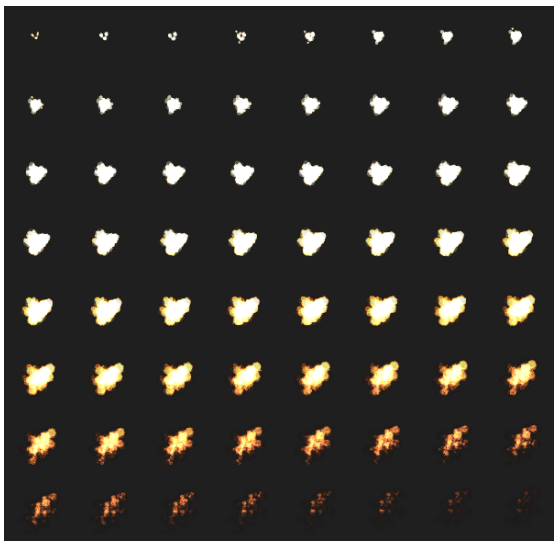

Figure 6: The texture used for rendering particles

identified using the lifeStage value passed at the shader, in order to do that, the vertex coordinate are computed with the code below (written in GLSL 1.5);

```
float row_texture = floor(lifeStage / 8);
float line_texture = mod(lifeStage, 8);
vec4 out_frag = texture2D( myTextureSampler, (UV / 8) +
    vec2((line_texture / 8), (row_texture / 8) ));
```

where:

- lifeStage: value passed from the application stage, see above for details.
- UV: initial texture coordinates of the square that cover the whole texture.
- myTextureSampler: GLSL sampler2D of the texture
- row_texture: integer that defines the row position in the map
- line_texture: integer that defines the line position in the map
- out_frag: output color of the shader.

In this way each time the lifeStage of the particle changes, the right sub texture is displayed changing its appearance from the initial spark (top left corner in the texture) to the exhausted flame (bottom right corner).

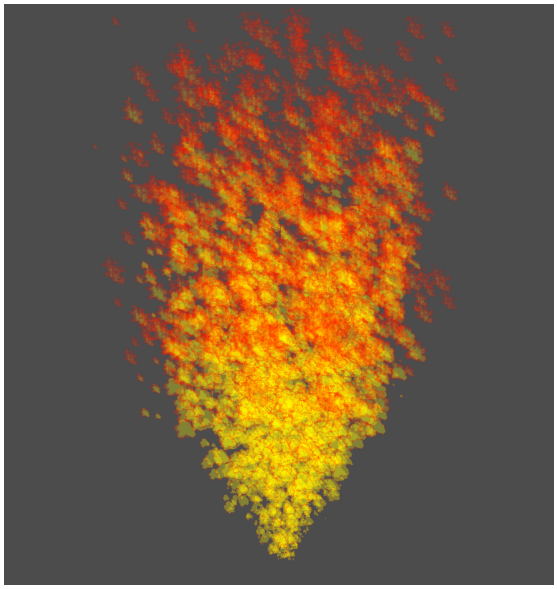

Figure 7: The final flame rendered

In order to increase the realism of the flames the opacity of each rendered texture has been reduced. In this way the composition of many particles has an

improved appearance. In Fig. 7 is displayed a frame of the flame implemented with this method.

6 Conclusions

Particle Systems are very useful to describe fuzzy phenomena in a realistic way. They allow to offer to the viewer sequences that appear correct and natural. Altough much research has been conducted on particle systems it is reasonable that even more of it will be carried out thanks to always better performing machines. The increasing computational power will unlock the possibility to model increasingly complex phenomena thanks to the usage of always more particles, able to describe these phenomena at even small granularity.

References

[1] William T. Reeves, *"Particle Systems- A Technique for Modeling a Class of Fuzzy Objects" ACM Transactions on Graphics*, Vol.2, No. 2, April 1983

[2] Karl Sims *"Particle Animation and Rendering Using Data Parallel Computation"*,Computer Graphics, Volume 24, Number 4, August 1990

[3] Lutz Latta, *"Building a million-particle system"*, GDC 2004

[4] Teng-See Loke *Computer Graphics and Applications, IEEE* Volume 12 , Issue 3, August 2002

[5] Niniane Wang, Bretton Wade *"Rendering Falling Rain and Snow"* ACM SIGGRAPH 2004 Sketches

[6] Wensheng Dong, Xinyan Zhang and Caijun Zhang *"Smoke Simulation Based on Particle System in Virtual Environments"*, 2010 International Conference on Multimedia Communications.

[7] Robert Bridson, Jim Hourihan, Marcus Nordenstam *"Curl-Noise for Procedural Fluid Flow"*, Proc. ACM SIGGRAPH 2007.