

3D video games

Particle Systems




Marco Tarini



1

Course Plan



- lec. 1: **Introduction** ●
- lec. 2: **Mathematics** for 3D Games ●●●●●●●
- lec. 3: **Scene Graph** ▶▶
- lec. 4: Game **3D Physics** ▶●●●● + ●●
- lec. 5: Game **Particle Systems** 📍
- lec. 6: Game **3D Models** ●●
- lec. 7: Game **Textures** ▶●●
- lec. 8: Game **Materials** ●
- lec. 9: Game **3D Animations** ▶●●●
- lec. 10: **3D Audio** for 3D Games ●
- lec. 11: **Networking** for 3D Games ●
- lec. 12: **Interactive Agents** for 3D Games ●
- lec. 13: **Rendering Techniques** for 3D Games ●

2

Particle effects

(aka «particle FX», «particle systems»)

- Digital representations of 3D objects...
 - Not easily described by their surfaces
 - And/or: very dynamic (variable topology)
- ...such as:
 - clouds, dust clouds
 - flames, explosions
 - water sprays, waterfalls, spouts
 - rain, falling snow
 - wind (transporting dust / leaves / etc)
 - steam whiffle, walking dust-puffs
 - custom visual effects (e.g. for magic spells, etc)
 - swarms of flies
 - sparks, fireworks, electric sparks
 - gusts of smoke
 - *and so on*



4

Particle effects: just a bunch of particles

- one particle represents
 - a water drop, a flame spark, a rain drop, a smoke puff...
- **state** of a particle
 - Newtonian state: position, velocity
 - maybe also : orientation, angular velocity
 - lifespan («time left to live»)
 - custom variables: size, color , etc...
- Each particle is
 - dynamically emitted, aka “spawned” (from an «**emitter**»)
 - evolved (state changes)
 - and disposed (removed), after a brief line

} according to
some predefined
criteria

5

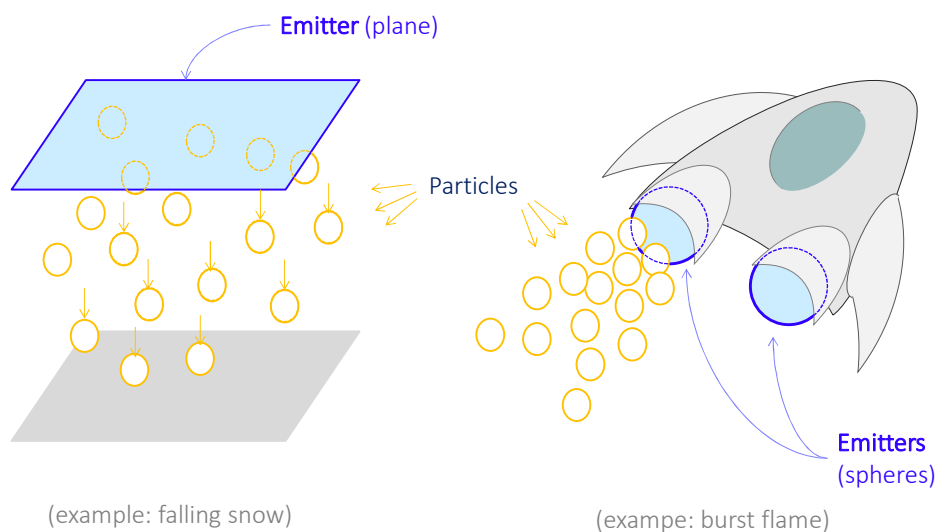
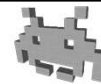
Particle effects: just a bunch of particles



- Particles of a particle system are a **simplified** version of particles in a physics engine
 - with much simplified: dynamics, collision handling
 - individual particles are not important!
 - it's the collective behavior (e.g. $10^1 - 10^6$ particles) that recreates the **visual** and the **behavior** of the recreated effect (flame, explosion)
 - the *entire* effect is often not that important either
 - cosmetics, not gameplay
- Note: particles systems are used in movies as well as videogames
 - We will discuss the videogame version

6

Emitters & Particles



7

Emitters: in the scene graph

- **Emitters** reside in a **scene graph** node
 - as such : it's positioned/oriented in the scene
 - as such : it has a local/global **transformation**
 - as such : is has its own local **object space**
 - to position/orientate the **emitter** *is* to position/orientate the **particle effect**

The blaze, the explosion, the spray of water, etc ...

8

Emitter: the producer of particles

- emits **particles** according a designated criterion...
 - in pseudo-random way
 - with chosen probability distribution
 - at a designated **rate**
 - how many particles/sec
 - produces particle with an initial state
 - initial pos: randomly generated inside **the emitter shape**
 - initial position (in emitter space), vel...
- ...for an established interval of time
 - e.g.: short (e.g. an explosion)
 - or medium (e.g. a blood gush from a wound)
 - or long (e.g. a column of smoke)
 - or undefined (e.g. water from tap, flame from torch...)

emitted particles per second

some function

time

9

Emitter's «shape»



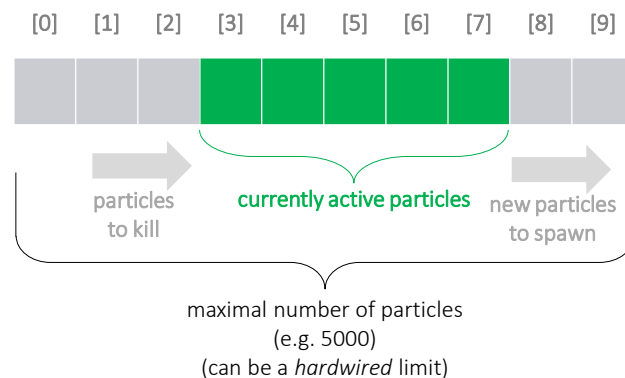
- An abstraction identifying the set of pos where new particles can be produced
- Just a 3D geometrical abstraction useful to guide particles creation
 - e.g. a sphere, cone, box, plane, point...
 - particle are created in a pseudo-random position inside this volume
 - Particle state: initialized with data expressed in **world space** or in **object space** (of the emitter)
 - e.g.: smoke: vel predominantly in Up dir. of *world space*
 - e.g.: rocket engine blaze: in Forward dir of *emitter space*

10

Data structure for running a particle system



- An array of particles
 - for each particle: its current status (position, velocity, time-to-live, ...)
- “Circular” array can be used



11

Internal data structure for a running particle system (pseudocode)



```
class Particle{
    vec3 pos;
    vec3 vel;
    float time_to_live; // seconds. how much longer?
    ...etc...
}

class ParticleSystem{
    Shape emitter;
    vector< Particle > particles; // circular array

    // interval of active particles
    int first_active, last_active;

    function evolve( float dt );
    function render();
    function init();
}
```

12

Particle systems: GPU implementations



- Running (i.e. “playing”, “executing”) a particle system is an extremely parallelizable task
 - especially if the used dynamics is simplified
 - each particles “evolves” on its own
 - spawn a “new” particle? Just reinitialize an existing particle at the initial state (circular vectors)
- GPU based implementations are relatively easy to do
 - GPU evolution
 - GPU rendering
 - particle data never leaves the GPU!

13

Particle systems: randomness / noise



- The spawning and evolution of particles typically use noise functions (pseudo randomness)
- Examples:
 - the initial position is randomly selected as any point inside the emitter
 - the initial color is selected as a random interpolation between two given colors
 - the speed and acceleration have random components
- This creates differentiation and reflect the stochastic nature of the simulated phenomena
 - Flames, etc

14

Evolution of the particles: simplified dynamics



Note:
Can be computed in: **emitter space**,
or **world space**, or **interpolations**

more procedural
(in the sense of a
simple procedure)




more
physically-based
(and expensive)

- Analytic evolution, kinematics
 - $state(t) \leftarrow f(t)$
 - we can edit the trajectory of the particle $f!$
 - kinematic particles – no real dynamics
- Numeric evolution, kinematics (no forces):
 - $state(t + dt) \leftarrow f(state(t), dt)$
 - not limited to real physics
 - e.g.: puff of smoke accelerate upward, snow falls slowly in zigzag way, water fall diagonally (due to wind), air bubbles in water accelerate upward + random
- Numeric evolution, dynamics (with forces):
 - give “mass” to particles
 - include (and cumulate) forces such as: cohesion between particles, repulsion between particles

16


Evolution of the particles: simplified collision detection



more procedural
(in the sense of a
simple procedure)


- No collisions!
 - e.g. smoke goes through walls (nobody cares)
 - easiest / fastest
- Collisions only with hardwired things
 - e.g., only with hardwired ground plane
 - still very easy to parallelize
- Collisions with all static objects in the scene
 - can use spatial indexing structure.
 - ponder for a given particle system: is this necessary?
- Collision with dynamic objects too
 - ponder for a given particle system: is this necessary?
- Collision with other particles too
 - luxury. Rare (in games)

more
physically-based
(and expensive)



18

Evolution of the particles: simplified collision response




more procedural
(in the sense of a
simple procedure)

If collision then...

- just kill the particle
- stop the particle: $vel = 0$
- *ad-hoc* changes in the particle state
 - e.g.: a water droplet just stops on a surface for a while (looks wet) then disappears
 - e.g.: in an explosion particles just becomes a black stain, stays for a while, then disappears
- full impact computation, but always **one-way**
 - elastic, static, or in between
 - particle is affected, object is not, even if dynamic
- full impact computation, possibly **two-ways**
 - the impacted object, if it's dynamic, is affected too
 - (rare, expensive)

more
physically-based
(and expensive)



20

Rendering a particle effect: way 1 – render each particle

Each particle is individually rendered, as...

- one rendering primitive
 - a point (“point splatting”), a segment...
- or, one small 3D model
 - few (or one!) polygons, maybe textured
- or, one *impostor*, i.e.
 - a small quad centered at the particle
 - oriented towards the observer (usually)
 - with a texture (often, animated: frames)
e.g. alpha maps + RGB maps
 - aka a “billboard”



popular solution

Final look = superposition of all particles

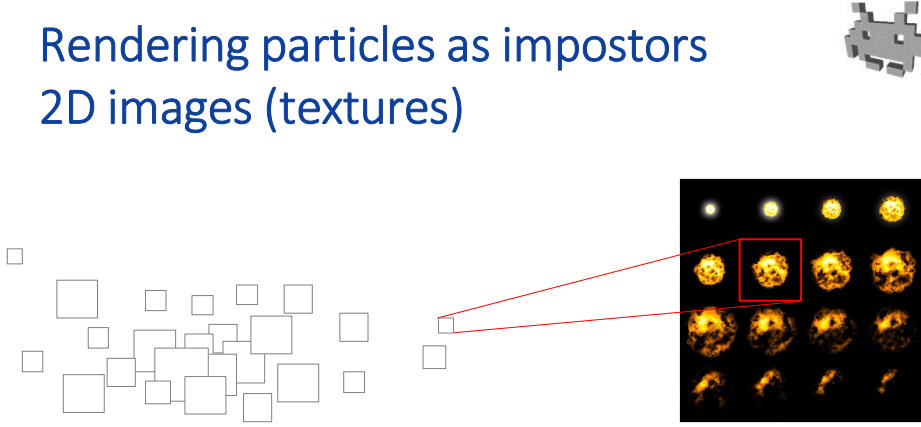
21

Rendering particles individually

- The aspect of individual particles is controllable in many ways
 - size of impostor?
 - color of the splat?
 - transparency level (alpha) the impostor?
 - screen-space rotation of the impostor?
 - if multiple sprites are available: which frame to use?
 - etc
- They can be parameters...
 - ...of time-to-live
 - e.g., for a flame: at start: red color; mid-life: yellow color; end: black color
 - e.g., for smoke:
at beginning small and dense particles; at end: large and transparent
 - ...of speed
 - ...or any other factor

22

Rendering particles as impostors 2D images (textures)

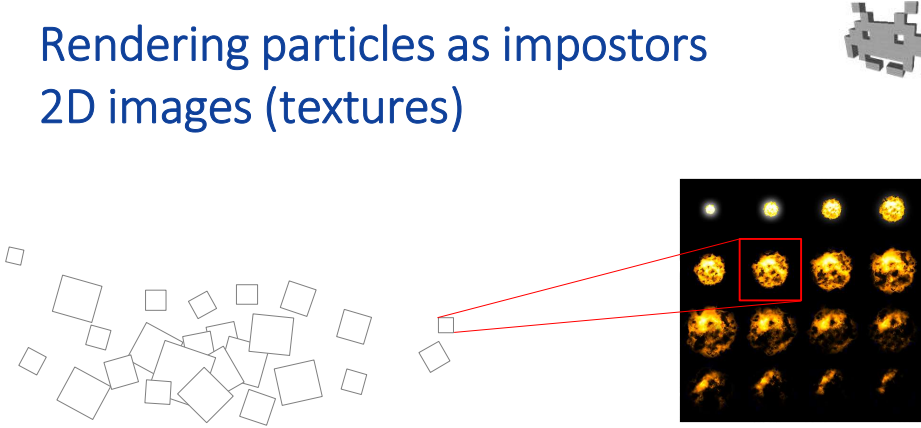


The image (aka sprite) can change during time
(animation, sequence of frames)

The image is partially transparent or semitransparent
(it has an "alpha" channel)

23

Rendering particles as impostors 2D images (textures)



can also be rotated in view space
(or, in 3D)

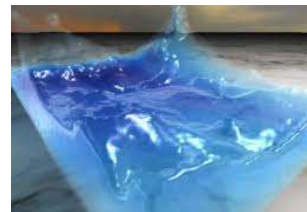
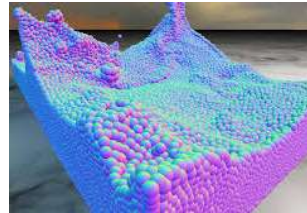
24

Rendering a particle effect: way 2 – fuse particles in one 3D shape



- Usually too time consuming, for a game
- Can be approximated with **screen-space techniques**
- pass 1:
splat a temporary “blob” for each particle in a offscreen buffer
- pass 2:
estimation of normals of “blobs” union in screen space
- pass 3:
rendering of the resulting surface
- Ideal for liquids!

see lecture
on Rendering later



this example by Simon Green (NVIDIA)

26

Authoring a particle effect



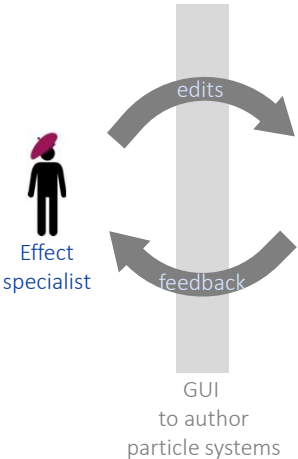
- Particle effect = just another asset
- Authoring it = the task of the *Effects specialist*
- Designing the **behavior**
 - define the emitter
 - specify how particles are **created** & **evolved**
 - how? by programming scripts for the task, or...
 - by specifying a predefined set of parameters through a GUI (in a particle systems authoring suite)
- Designing the **look**
 - which **image (texture)** for **impostor**
 - which tiny **3D models** ?
 - which **splat** parameters, etc.



digital
artist

27

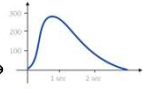
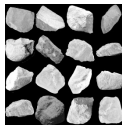
Authoring a particle effect via a GUI



Effect specialist

GUI to author particle systems

Particle Effect asset

- Spawning parameters
 - emitter shape
 - emission ratio (over time) → 
 - initial state of particle (e.g. velocity)
 - initial time to live
- Evolution parameters
 - particle trajectory
 - changes in vel
 - forces, etc
- Rendering parameters
 - Rendering strategy
 - Colors
 - Sprites / textures → 
 - Used 3D model, etc.

All that as a function of time, or as a distribution random variables...

28

Many particle effect framework / software exists

Example of specialized tools:


- Houdini (widely used for movies)
- Cascade (in Unreal)
- Particle Flows (in 3D studio Max)
- X-Particles (for Cinema4D)
- thinkingParticles (plug-in for different software)
- ...and many others

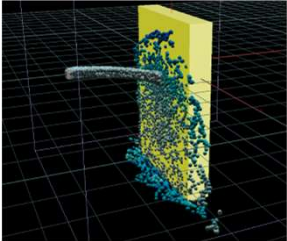
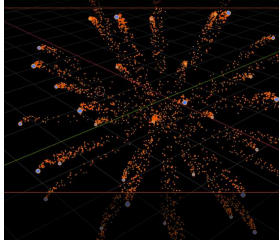


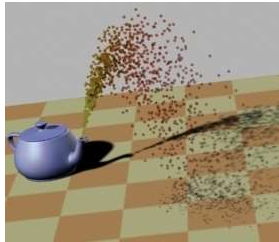
Many systems provide their own built-in editors

- Unity ("shuriken") wysiwyg slider-based editor
- Blender
- Maya ("nParticles")
- ...and many others

29


Particle effects in...




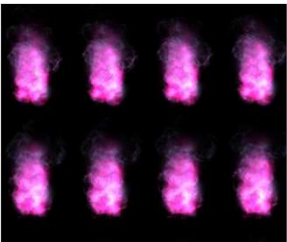




		
Houdini	nParticles (Maya)	Blender
		
Cascade (Unreal)	Shuriken (Unity)	RenderMan 20

30

Particle effects in...



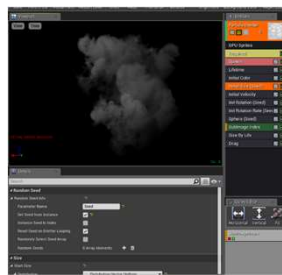
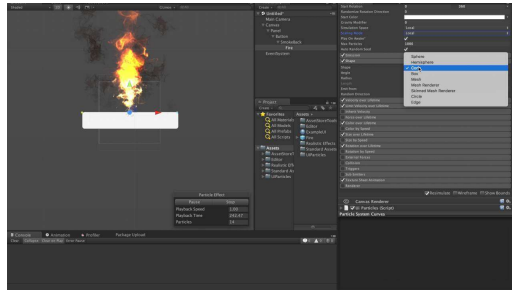
		
Particle Flow (3D max)	X-Particles (Cinema4D)	Thinking Particles
		
TimeLineFx (RigzSoft)	PocCornFX	Particle Illusions (Boris FX)

31

Just two notable examples




- Unity built-in editor for “shuriken” particle systems
- Unreal built-in editor for “cascade” particle system



32

Lack of established formats for particle-effect assets



- Each software suit uses its own:
 - set of **parameters**, tricks, degrees of **customizability**
 - interface to let a **FX specialist** author the particle system
- ...and file formats to store that **asset**. Examples:
 - Unity: stored as .prefabs
 - Unreal: “cascade” file format
 - Maya: .pdb .pda
 - Renderman: .ptc
 -  Houdini: .geo .bgeo

33

Lack of established formats for particle effect assets



- Problems:
 - hard to **run** a particle system in a game engine unless that particle system was **authored** in that engine/system
 - hard to reuse or off-source particle systems across different systems / engines
- To solve this, a few “Esperanto” format have been proposed for particle systems:
 - still not very established



(by Disney)



ALEMBIC

(by Sony)

34

Particle effect: cosmetics or gameplay?



- Typically, it's only graphic coating
 - known to increase visual realism / immersion
 - communicates what's going on to the player (e.g., splashes = “you are walking on water”. metal sparkles = “you have been it”)
 - gameplay not affected
 - this justifies many approximations
- Remarkable exceptions exist
 - particles affecting gameplay

Portals, Valve, 2011



36

Digression: particle effects outside videogames

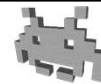


- Particle effects are used in **movies** too
 - the techniques are the same
 - naturally, there is less need for **simplification**
 - intended for **off-line** rendering not **real time**
 - a few of the sw tools listed above are specialized for this scenario
- Additional use of particle systems in movies: **fur / hair / grass**.
 - imagine the trajectory of each particle as shape of an individual hair instead of the position as a function of time



38

Practical (and fun) exercises



- Improvise yourself as a *FX specialist*
 - use any of the above software (e.g., Unity or Unreal)
 - use its interface to create a particle system to simulate ... something (an explosion, a gush of water)
 - maybe follow some tutorial
- Observe some existing particle effect
 - download them from repository / asset stores
 - analyze them from the interface
- *Reminder*: this course is does not cover any digital artist's skills, but experimenting always helps you familiarize with the process of *asset creation*

39