EDAN35: A Snow Particle System

Johan Pettersson* Dennis Jin†

Lund University
Sweden

Abstract

In this paper, we describe the implementation of a particle system intended for approximating and displaying snow and other forms of precipitation such as rain and hail. The results are showcased, performance measurements are given, and some possible improvements are discussed.

Figure 1: Snow in the Sponza atrium scene.

1 Introduction

Particle systems are ubiquitous in many modern visual applications: they are used for common effects such as explosions, magic, weather, and much more. They can be implemented so that most of the particle update logic is computed on the CPU, or alternatively on the GPU. The latter allows for far higher performance with greater numbers of particles due to avoiding recurring memory transfers of particle state from CPU to GPU, and state update computation on the CPU; however, a disadvantage is the complication of access to sparse representations of the scene for e.g. collisions between particles and scene geometry, and necessitating the use of buffers of fixed size for all particle state.

In this implementation a non-screenspace solution for collisions was implemented, allowing for particles to fall and collide with the scene outside of the view frustum. The test scene was the Sponza atrium from Crytek, chosen because of its reasonable geometric complexity suited for demonstrating the scene geometry collision detection.

2 Algorithms

2.1 GPU-side particle updates

Transform feedback is used to update the particles entirely on the GPU, in conjunction with a geometry shader to enable conditional discarding of particles beyond the bounds of the scene (as determined by the CPU code and passed to the shader in uniforms) and particles that have collided with scene geometry and been at rest for a certain length of time.

Each particle stores its position, velocity, lifetime (the time since the particle was spawned), and type (used only for distinguishing between inactive particles that have collided with the scene geometry and those that haven't). On each frame, every moving particle's velocity v is updated by adding the contributions from several forces (gravity, drag force (proportional to speed squared), and wind), and the position is updated to $p_t = p_{t-1} + \Delta v$, where Δ is the time since last frame.

2.2 Collisions

At program startup, the CPU code computes a collision octree for the scene, encoding whether or not each node is intersected by triangles down to a chosen octree depth (set to 10 for the Sponza scene loaded by the demo, using Tomas Akenine-Möller's triangle-box overlap testing code [Akenine-Möller 2001]). This sparse octree is then used to construct a non-sparse 3D texture that encodes the empty-or-intersected status of each volumetric element in the scene; the particle update geometry shader then samples this texture at the location of each particle, stopping the particles that are in volumetric elements intersecting the scene triangles.

2.3 Motion blur

In reality, both cameras and the human eye do not usually sample incoming light from just an infinitesimally short span of time; rather, they sample continuously, an average over some time. Due to this, fast-moving objects (relative to the camera/eye) appear to streak in the direction of motion: this effect is known as motion blur and is required for "realistic" depiction of fast-moving objects.

In this particle system, the render geometry shader approximates a particle's position in the last frame as its current velocity multiplied by the frame time delta subtracted from the current position, and stretches the particle so that it covers the intervening space; this is a crude approximation of real motion blur but provides an improved sense of the direction and speed of motion.

3 Results

3.1 Memory requirements

The collision texture requires $\frac{\text{res}^3}{8}$ bytes of storage, where res is the resolution (a single bit is sufficient for coding empty/non-empty, so eight nodes fit in one byte). At res $= 10$, this is 128 MiB; accuracy can be traded for memory by lowering the resolution.

A single particle stores its position, velocity, lifetime and type; this is eight single-precision floating point numbers in total, meaning a single particle requires 32 bytes. The particle buffers are allocated once with a fixed maximum size; at a size of 2 million, the two buffers combined require 64 MiB.

3.2 Performance

Due to some inefficiency in the CPU-side Bonobo code, the largest part of each frame is spent issuing the scene geometry draw calls. Disabling V-sync and scene geometry rendering allowed the measuring of an upper bound on the time spent on the particle system; running at 2560x1440 resolution on an Intel i7 3770k CPU and an NVIDIA GTX 1080 Ti GPU, the following upper bound times were measured for different particles-per-second spawn rates:

*e-mail: dat14jpe@student.lu.se
†e-mail: dat14dji@student.lu.se

Figure 2: Fast-moving (sideways, as seen from this perspective) snow particles.

Figure 3: From left to right: 1/10 exposure; normal exposure; 10 times exposure.

Spawn rate	Frame time	FPS (= 1000 / frame time)
1000/s	0.8 ms	1200
10 000/s	0.8 ms	1200
100 000/s	1.1 ms	900
300 000/s	2.4 ms	410

(no significant difference between the first two measurements was recorded)

4 Discussion

4.1 Update

The particle system can be extended to work in a large world: it is computationally prohibitive to simulate in detail (that is, per-particle) areas much larger than the Sponza scene. The particle system can therefore be re-centered around the current camera location on each frame, with a fixed extent beyond which particles would be discarded (as they move farther away from the camera than the fixed extent, rather than beyond the entirety of the world). A problem with this is determining where to spawn new particles and with which initial state: unless indoor precipitation is acceptable, particles can no longer simply start at the top of the scene and be allowed to fall all the way down, but instead, particles must appear to have been in existence for a long time before they were actually spawned. Determining possible spawn locations for new particles can be done on the CPU.

The collision texture required for each scene could be precomputed to minimize runtime overhead; another option would be to use a GPU voxelization algorithm, or perhaps reusing voxel data from another part of the rendering (such as voxel-based global illumination, if present).

Figure 4: High terminal velocity, low alpha and low size result in a look closer to rain.

4.2 Render

As for the rendering, the particles in this implementation aren't affected by lights; the quality can be improved by rendering them back-to-front (for correct blending), and using a clustered deferred rendering system to take into account mostly only the light sources affecting each particle. For correct back-to-front rendering, the particles have to be sorted by distance; this can be accomplished by implementing a variant of radix sort (or perhaps bitonic merge sort [Peter Kipfer 2005], if the former proves too slow).

The motion blur approximation isn't very accurate: when using actual frame times instead of fixed times, slight delays cause noticeable "flashing" as particles briefly gain and quickly lose brightness when stretched more. Correcting this (by integrating along the direction of motion instead of using ad hoc lengthening of particles) would also enable the use of proper motion blur from camera movement (which is otherwise likewise affected by the "flashing" artifact).

Textures could potentially be used to improve the look of the particles, but there could be a risk of the "flatness" being visibly distracting; perhaps 3D textures can be utilized, with the per-particle type parameter encoding which texture(s) to look up (and blend between) among multiple textures for different particle types.

4.3 Suggested directions

The per-particle type parameter can be used to differentiate between many types of particles; different per-particle terminal velocities, for instance, could allow for more interesting patterns of motion in a group of particles.

Inactive particles (those that have collided with the scene geometry and now have zero velocity) simply fade out over a short length of time in this implementation; in reality, snow (and water) should instead accumulate. This could be simulated in real-time, possibly by adding inactive particles to a volumetric structure representing the accumulation (perhaps in a real-time adaptation of the material point method described in a Disney paper [Alexey Stomakhin 2013]).

As seen in Figure 5, abstract scenes can be rendered by the system as-is; the system can be extended to have many different colors and properties per-particle, not necessarily constrained by or even based on real physics, to try to produce varied fractal-like patterns.

Figure 5: Pausing the particle updates while using extreme exposure and wind strength (and disabling the G-buffer geometry render and deferred shading lights) can create interesting abstract patterns.

Figure 6: Only particles (no scene geometry) rendered, after an extremely strong burst of wind (created by manually adjusting the wind strength).

References

AKENINE-MÖLLER, T. 2001. Fast 3D Triangle-Box Overlap Testing.

ALEXEY STOMAKHIN, CRAIG SCHROEDER, L. C. J. T. A. S. 2013. A material point method for snow simulation.

PETER KIPFER, R. W. 2005. *GPUGems 2*. ch. 46. Improved GPU sorting.