1. Introduction

Particle systems are used in computer graphics to model a class of fuzzy objects such as fire, clouds and water. Each particle of the system can be assigned its own range of motion, and behavior.

In this assignment you will use a particle system to create a “cool” animation, e.g. an erupting volcano, waterfall, etc. You will receive credits based on your animation's artistic and technical merit.

Although the particle system you will implement is rather basic (e.g. no inter-particle interactions), you will make use of standard graphics techniques such as collision detection and handling, texturing, shading and blending effects to improve the realism of your rendering.

Some additional resources you may find useful for completing the project are listed below:

- Nvidia's “cloth” animation: [http://www.win.tue.nl/~wstahw/2IV40/cloth.tgz](http://www.win.tue.nl/~wstahw/2IV40/cloth.tgz)

![Fig. 1: Karl Sims waterfall.](image-url)
2. Assignment

The best way to complete the assignment is to start from an already working program (e.g. Nvidia's cloth demo), modify it to implement certain functionality test it, and then move on to the next step. The cloth demo already contains some useful pieces of code such as p-buffer manipulation, render to texture (via copying from p-buffer to the texture) and render to vertex arrays. In principle, the provided (low-level) functionality should be enough such that you can implement the assignment.

You must write the code to manage features of the particle system (e.g., create, delete particles, compute forces, collisions, time integration), perform all rendering and handle any other additional functionality you may desire.

2.1 The volcano/waterfall and terrain

To render the volcano/waterfall and the surrounding terrain you will need to generate a height field using some image manipulation software (GIMP, Photoshop, etc.). Store the height field as a gray scale image, which you can easily load and map to a texture. The terrain and volcano/waterfall can be rendered as a simple triangular mesh, generated using the height field. The user should be able to manipulate the camera using the mouse to spin and move the height field. You could also implement a sky box and use some suitable texture on the terrain to improve realism. The height field should also be stored into a texture useful to constrain the particles to stay on the surface of the terrain.

2.2 A basic particle system

The next step is to implement a basic particle system. Each particle of the basic system should have some attributes such as position, velocity, age, size, etc. For each frame of an animation sequence the following steps are performed (not necessarily in this order):

1. New particles are generated.
2. Each new particle is assigned its own set of attributes.
3. Any particles that have existed for a predetermined time are destroyed.
4. The remaining particles are transformed and moved according to their dynamic attributes.
5. An image of the remaining particles is rendered.

Particle positions and velocities should be stored in textures such that you can update these in a fragment program. Particle sizes can be generated on-the-fly in a vertex program and should be varied as a function of their ages. The other particle attributes can be conveniently stored as vertex attributes which are then accessible to your shaders when rendering the particle system using OpenGL vertex arrays. For now, render the particles using point primitives of a given size. Also, have particles interacting with gravitational forces, the height field and other collision obstacles you chose to implement, and an “interesting” force field (e.g. use some ideas from Karl Sims' paper). Allow in your animation for a maximum number of particles; when the maximum number of particles is reached, you may reset or stop the animation.

2.2 Adding and removing particles

In this stage you will implement functionality to add new particles and remove old ones. As some required functionality may not be available on your GPU, creation and deletion of particles is mostly done on the CPU. For each active particle on the GPU, you should store on the CPU side some
information related to the lifetime and age of that particle. For example, you may store for each particle its death time and index into the 2D texture storing its velocity and position. Then, for each particle, at each timestep (or number of steps) of the animation you should check if its death time is larger than the current animation time. If this is the case, store its index into an array and remove the particle from the list of active particles. The places of the death particles can now be taken by newly generated particles. At each time step an emitter generates new particles (if slots are available) with a given, user-adjustable rate. On the CPU, newly generated particles are represented by position, velocity, index into position and velocity textures, and data about the age of the particle. Once new particles are generated on the CPU, their velocities and positions should be made available also for the GPU, i.e. they should be stored in the corresponding textures. This can be done by rendering points with coordinates representing the index of an empty slot and texture coordinates containing the initial 3D position of the particle. Then a simple fragment program outputs as color the texture coordinates representing the desired particle position. The same process should be used to transfer initial particle velocities. Finally, the vertex array containing other GPU-side particle data (e.g. death time, birth time, color, etc.) has to be updated by mapping the GPU memory on the CPU and updating the corresponding memory locations.

### 2.3 Improved particle system (I)

**improvement I: 10 points**

Since now the particle system is fully functional, you should move on and focus on improving its appearance. The first step towards achieving this is to render the particles using point sprites. This basically means loading an appropriate texture representing a spherical object as a height field, enabling the point-sprite extension and alpha blending, and setting up a fragment program which computes the final particle color by modulating the color of the particle with that extracted from the sprite texture. Implement also motion blurring based on the previous and current particle positions.

### 2.4 Improved particle system (II)

**improvement II: 30 points**

Finally improve the appearance of the particle system using shading. Here, you'll have to be creative to integrate shading in your animation. Some useful ideas you may find in the paper by van der Laan et al., “Screen space fluid rendering with curvature flow” available here: