Particle Simulation

1 Overview

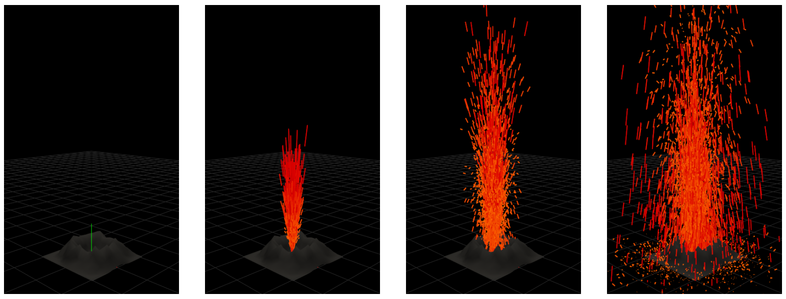

Figure 1: Progression of the lava particle simulation

Particles are ubiquitous in the field of computer graphics, and have been used to simulate many complex physical effects. One of the most common areas of use is special effects, such as sparks and explosions. In fact, the use of particles extends far beyond the domain of special effects. In fact, fluids, fire, water, smoke, cloth, and even physical character animation has been done through a particle representation. Luckily for us, not only is particle simulation very useful, but it is also one of the simplest mechanical systems to implement. For this lab, you will be using particles to simulate the appearance and motion of lava.

2 Particle Dynamics

Let x, $\dot{x}$, and $\ddot{x}$ denote the position, velocity and acceleration of the particle. Newtons second law of motion is given by the familiar equation:

$$f(x(t), \dot{x}(t), t) = m\ddot{x}(t) \tag{1}$$

where f, m, t are the force, mass of the particle, and the time. To determine the quantity of interest, x, at some time t, we must integrate this second-order ordinary differential equation (ODE). Depending on the form of f, which we will assume to be non-linear in its inputs, it may not be possible to solve Eqn. 1 analytically. Instead, we will approximately solve Eqn. 1 using numerical integration.

To perform numerical integration we must first convert Eqn. 1 from a second-order ODE into a system of first-order ODEs. In general, we can convert any n-th order degree ODE into a system of n first order ODEs by introducing new variables and substitution. The converted system from Eqn. 1 is given by:

$$\begin{aligned} \dot{x} &= v \quad (2) \\ \dot{v} &= \frac{f}{m} \, . \quad (3) \end{aligned}$$

Here, we have introduced a new variable, v, and substituted it for $\dot{x}$, the velocity of the system to remove $\ddot{x}$. Note that the system is still dependent on f, which we've assumed is a non-linear function; thus, we have a non-linear system of first-order ODEs. Therefore, we cannot apply analytic methods for solving first-order linear ODEs to this problem; we will resort to numerical integration instead.

3 Numerical Integration

Numerical integration works updating a state vector iteratively, using a numeric integration formula, to advance the system in time. Typically, we wish to know the position and velocity of the particle at some given time so that we may render it. To find these quantities of interest, we must first initialize our system to some given state, $[x_0, v_0]^T$, which is the initial position and velocity of the particle at time $t = 0$. Each iteration step we advance time by h units, the integration step size, and update the state using the integration equations: thus, $t = ih$, where i is the ith integration iteration.

There are many different techniques for integration. We will update the state using the simplest numerical integration technique, explicit Euler integration. Explicit Euler integration is given by the following pair of update rules:

$$\begin{aligned} x_{i+1} &= x_i + hv_i \quad (4) \\ v_{i+1} &= v_i + \frac{h}{m} f_i \ . \quad (5) \end{aligned}$$

which are generated through the approximation $\dot{x}_{i+1} \approx \frac{x_{i+1}-x_i}{h}$ and $\dot{v}_{i+1} \approx \frac{v_{i+1}-v_i}{h}$ plugged into Eqn. 2 and 3. From calculus, we know that as h is increased, the approximation becomes worse. In other words, continuous differentiable functions are locally linear.

Explicit Euler integration works by first updating the current position of the particle using the velocity of the previous timestep scaled by the step size. Then the velocity is updated using the current forces applied to particle scaled by the step size. In general, the forces can come from constraints between the particles, gravity, wind, friction, and other force fields.

It's important to note that h is tied to the accuracy and stability of the numerical integrator. Stability here refers to integration error manifesting itself in the form of energy gain which may eventually result in the system exploding (for this lab you will not introduce any force sources which should cause this to happen). Smaller steps typically result in more physically accurate and stable solutions, but require more iterations to integrate. For example, if we wanted to integrate our particle system for $1s$ of time with a step size of .001, we would need to perform 1000 integration steps. That would be acceptable for a single particle, but would probably be too slow for the interactive simulation of 100,000 particles.

There are many different numerical integration techniques that exist with better stability and accuracy than explicit Euler integration; although, none of them are simpler to implement and understand. In practice, if you were implementing a particle system to simulate cloth or fluids, simulations which exhibit large constraint forces between particles, you would probably want an integrator with better stability and accuracy, such as Verlet integration.

4 Force Sources

Up to this point we havee discussed how to integrate the state of the system provided forces, but not how these forces are generated. For this assignment you, will have two sources of force: gravity, f_g, and the ground. As mentioned previously, many additional interesting sources of force can be introduced (in some cloth simulations, cloth is simulated using spring-mass systems consisting of particles attached together in a lattice network of springs).

During each integration step you will need to add the gravity force to the particle:

$$f_g = [0, -mg, 0]^T \ , \quad (6)$$

where g is the gravitational constant ($g = 9.81$ for Earth). This will cause your particles to fall towards the ground plane. If you detect the height of a particle, $x_y < 0$, rather than applying a force which will not immediately correct for the interpenetration, you will apply an impulse, a force applied over an infinitesimal duration to correct for penetration immediately. Sounds complicated? It's not. If $x_y < 0$ set $x_y = 0$ and then update the y-component of the velocity using Poisson's restitution rule:

$$v_y^+ = -\epsilon v_y^- \tag{7}$$

where $\epsilon \in (0, 1)$ is a restitution coefficient which controls the bounciness of the particle, and v_y^+, v_y^- are the post-impact, pre-impact velocity along the y-axis. $\epsilon = 0$ generates a perfectly inelastic collision with the ground, in which all kinetic energy is dissipated, while $\epsilon = 1$ corresponds to a completely elastic collision, in which all kinetic energy is preserved. You've effectively introduced a force which is applied over an infinitesimal duration of time. This impulse rule will guarantee that your particle does not pass through the ground plane.

In addition to restitution, you will also want to apply damping forces to remove the tangential velocity of the particles, so that they do not slide across the ground indefinitely. If $v_y < 0$, update $v_x = \alpha v_x$ and $v_y = \alpha v_y$, where $\alpha \in (0, 1)$. α controls how much tangential kinetic energy is conserved during each impact. Higher values result in a more slippery surface.[1]

5 Rendering

There are many ways to render your particles. For example, if you were performing a fluid simulation, you could construct a continuous surface around the particles to represent the surface of the water. If instead, you were doing a cloth simulation, the particles would be vertices of a mesh representing the cloth, attached together with springs. For this lab, we will be rendering the lava using simple lines.

For each particle, you will want to draw a line between position of the particle, and some other point displaced in the direction of velocity of the particle:

$$\begin{aligned} p_1 &= x \quad (8)\\ p_2 &= x + sv\ , \quad (9)\end{aligned}$$

where p_1 and p_2 are line endpoints, and s is the length of the line relative to the velocity ($s = .04$ works well). This will create a line in the direction of motion with a length that is proportional to the velocity of the particle.

To create the appearance of lava, you can set the color of line based upon the velocity. In Figure 1, particles which are traveling slow are given the color orange, and particles which are traveling fast are given the color red. This simulates the cooling of the lava as it flies through the air. You might also want to make the color depending upon time since the particle was emitted from the crater.

6 Implementation Notes

You should create a struct containing the particles position, velocity, mass, applied forces, and color. You'll then want to create an array of several thousand particles (about 5000 should do). Your lava crater should emit particles at some frequency (about 1000 per second) indefinitely. Since you only have a finite number of particles, you'll want to reuse the oldest particles emitted. This can be done easily by treating your array as a ring buffer and simply looping around, re-emitting particles which are pointed to by the current ring index.

You will also want to initialize the particles at some location, with some velocity. In general, to create a convincing lava effect, you will need to randomize the starting positions and velocities for each particle a little bit. Randomize separately for each velocity element.

[1] This is technically not the best way to model kinetic friction, but for this assignment, it will do an admiral job