Project A: Independent Particle Systems’ Interactions Modeled Realistically.

Michael Huyler (msh1851)

15 February 2020

# User Guide

## Goals of this Program

This project aims to serve as a visual representation of four different types of particle systems, implemented in JavaScript and WebGL. The four particle systems (in order from left to right, top to bottom as seen in Figure 1) are as follows.

1. 90 “boids,” or flocking particles, modeled using the methods described by Reynolds (1986).
2. 2000 flame/fire particles, modeled using the methods described by Reeves et al. (1983).
3. 300 particles linked together by springs to form a simulated cloth.
4. 600 particles constrained in a volume, under the effects of a vector field.

Each particle system demonstrates interesting particle interactions, and the unique features of each are listed below.

1. The boids particles will “wrap around” their container in the x and y axes while maintaining a flock-like cohesion, and actively avoiding a “predator” obstacle that travels randomly.
2. The particles that make up the flame are emitted with even distribution across the entire surface of the sphere that is located within the bounds of their container.
3. The springs between particles are visualizable, as is the complex geometry the cloth falls on. The springs change color from white to blue to red as their level of deformation increases.
4. The particles simulate snowfall, as the flakes fall to due to gravity, they are blown into a vortex before eventually landing. After landing, they will melt, and new snow will fall to replace them.

## Instructions on Using this Program

Upon loading the file HuylerMichael\_ProjA.html, the user need not do anything in order to observe the particle systems. However, their interactions become more interesting when the user interacts with the program. Listed below is a complete list of actions the user can perform in order to manipulate the program in some way.

Actions that are performed immediately upon their corresponding keys being press.

|  |  |
| --- | --- |
| . | Toggles the on-screen help menu, which lists all actions available to the user. |
| / | Toggles the on-screen GUI, which allows the user to perform specific manipulations to particle systems. |
| P | Pauses and unpauses all particle systems (the camera remains moveable when paused). |
| <Up Arrow Key> | Increases simulation speed (capped at 60 FPS). |
| <Down Arrow Key> | Decreases simulation speed (bottoms out at approximately 4 FPS). |
| <Space> | Resets and drops the cloth. Cycles forces applied to snowfall. |

Actions that are performed continuously when their corresponding keys are held.

|  |  |
| --- | --- |
| W, A, S, D | Moves the global camera. |
| I, J, K, L | Aims the global camera. |
| O | Enables “slo-mo” mode. |
| C | Prevents screen clearing, creating the illusion of “trails” left by each particle. |

Actions that are performed with the mouse.

|  |  |
| --- | --- |
| <Left Mouse Button> | Wiggling the mouse with this button held will cause the vortex in the snow particle system to increase in strength. |

It should be noted that all actions can be performed concurrently, allowing the user to, for example, rotate clockwise around a fixed point by strafing with the <A> key and rotating with the <L> key, while holding the <O> key to observe the particles at ¼ speed.

Furthermore, the program state can be altered in many interesting ways using the on-screen GUI, including changing the solver (with both explicit and implicit solvers selectable), toggling constraint visibility, and even directly manipulating constraint positions and sizes.

# Code Guide

HuylerMichael\_ProjA.html

This HTML file sets up the canvas that will contain the WebGL context.

style.css

This CSS file styles menus.

HuylerMichael\_ProjA.js

This JavaScript file sets up the main control loop, initializing all the particle systems, forces, constraints, and VBO boxes (described below).

PartSys.js

This JavaScript file describes the Particle System class. To instantiate a Particle System, the constructor only requires a particle count. There are four types of Particle Systems: Snow, Cloth, Boids, and Fire. A Particle System has the following public properties:

type  
The type of particle system to simulate.

s0, s1, s2, sM, s3, sErr  
State arrays (previous, current, next, midpoint, 2 states ahead, and approximate error, respectively) describing each particle’s position, velocity, acceleration, color, mass, radius, and age.

s0dot, s1dot, s2dot, sMdot  
The derivative of the corresponding state array with respect to time.

force\_set  
The set of forces acting on this particle system.

constraint\_set  
The set of constraints enforced on a subset of the particles in this particle system.

A Particle System has the following public methods:

void init(part\_system\_type, my\_vbo, constraint\_vbo, force\_set, constraint\_set, initial\_conditions?)  
Takes a PARTICLE\_SYSTEM type, the index of the VBO that will handle drawing particles, the index of the VBO that will handle drawing constraints, the set of forces that will act upon particles, the set of constraints that will be enforced on particles, and an optional set of initial conditions with which the Particle System can be initialized. Initializes this Particle System before the main loop begins.

void blink(state?)  
Takes an optional state. If the state is nonnull, it is applied to the particle system on the current frame.

void applyAllForces(state)  
Takes a state array and passes it to the apply(state) method of all the currently enabled forces.

void dotFinder(state)  
Takes a state array and computes the derivative of it with respect to time.

void solver(solver\_type)  
Takes a SOLVER type and uses it to compute the next state of the particle system.

void doConstraints()  
Enforces all currently enabled constraints on this Particle System.

void render(index?)  
Takes an optional index where VBO contents updates should begin. Renders this Particle System based on the updated state each frame.

void swap()  
Propagates state arrays backwards, setting s0 to the contents of s1, and s1 to the contents of s2 for use in future timesteps.

void enableForce(force)  
Takes the index of the force to be enabled.

void disableForce(force)  
Takes the index of the force to be disabled.

void enableConstraint(constraint)  
Takes the index of the constraint to be enabled.

void disableConstraint(constraint)  
Takes the index of the constraint to be disabled.

string toString()  
Returns a string representation of this Particle System.

void insertGui()  
Generates folders and settings in the on-screen GUI based on the contents of this Particle System.

Force.js

This JavaScript file describes the Force class. To instantiate a Force, the constructor requires a FORCE\_TYPE and a list of affected particles. The types of Forces implemented in this program are Simple Gravity, Drag, Wind (Vector), Spring, Flocking (Boid), Line Attractor, Uniform Point Attractor (constant attraction), Point Attractor (attraction diminishes over distance), and Vortex. Depending on which type a Force is, it will have different properties. For Vector Forces (Simple Gravity, Drag and Wind), these properties are the *x*, *y*, and *z* components of the vector describing the force, and the *magnitude* of the vector. For Spring Forces these are the spring constant *k*, the natural length of the spring *lr*, and the damping constant *d* of the spring. For Flocking Forces, these are the *minimum radius*, *maximum radius*, *binocular angle*, *monocular angle*, *avoidance*, *velocity matching*, and *centering*. The last three are unitless hyperparameters. For Attractor Forces (Line, Uniform Point, Point, Vortex), these are the *x*, *y*, and *z* components of the position of the force, the *x*, *y*, and *z* components of unit vector in the direction of the force, the exponential “tightness” *p*, the length of influence of the Force *L* (for Line Forces only), and the radius of influence of the Force *r* (for Vortex Forces only). All Forces also have a *type* and a list of particles they affect. All Forces have the following public methods:

void enable()  
Enables this Force.

void disable()  
Disables this Force.

void apply(state)  
Takes a state array and applies this force to all particles in the state array that are listed in this Force’s particle list.

void draw(vbo, index, enabled, p0, p1)  
Takes a VBOBox that will handle drawing this Force, the index of this Force, whether this Force is enabled (and thus drawn), and two points. Draws this Force on screen. This is only used for Spring Forces, whose visualization is the most helpful. The Force is drawn as a line between the two point parameters.

Constraint.js

This JavaScript file describes the Constraint class. To instantiate a Constraint, the constructor requires a CONSTRAINT\_TYPE, a list of affected particles, an optional flag for enabling particular walls, the restitution of the constraint, and a list of bounds. The types of Constraints implemented in this program are Rectangular Volume Enclosure (Impulsive, Velocity Reverse, and Wrap-around [hitting a wall “wraps” the particle to the opposite wall]), Rectangular Volume Obstacle, Sphere Obstacle, and Absolute Position. Depending on which type a Constraint is, it will have different properties. For Rectangular Volume Constraints (Enclosures and Obstacles), these are the maximum and minimum bounds in the x, y, and z directions, and the restitution. For Spherical Constraints, these are the position and radius of the sphere, and the restitution. For Absolute Positioning Constraints, these are simply *x*, *y*, and *z* position. All Constraints also have a *type* and a list of particles they are enforced upon. All Constraints have the following public methods:

void enable()  
Enables this Constraint.

void disable()  
Disables this Constraint.

void constrain(current\_state, next\_state)  
Takes two state arrays and enforces this Constraint on all particles in the next state array that are listed in this Constraint’s particle list.

void draw(vbo, visible, r, g, b)  
Takes a VBOBox that will handle drawing this Constraint, whether this Constraint is visible, and a color. Draws this Constraint on screen.

string toString()  
Returns a string representation of this Constraint.

GUI.js

This JavaScript file sets up the on-screen GUI using the dat.gui library, and handles toggling the GUI and the on-screen help menu.

VBOBox.js

This JavaScript file describes the VBOBox class. The workings of this file are not relevant to this project and will thus be omitted. The VBOBox class is inspired by the work of Professor Tumblin on his own VBOBox prototype, restructured to better suit the needs of this program.

Miscellaneous

The following JavaScript files are libraries provided freely online. Their use in this project is permitted by their respective licenses, and credit is given to their respective authors in the header docstrings of each file.

cuon-matrix-quat03.js  
cuon-utils.js  
dat.gui.min.js  
gl-matrix.js  
sha1-min.js

# Results

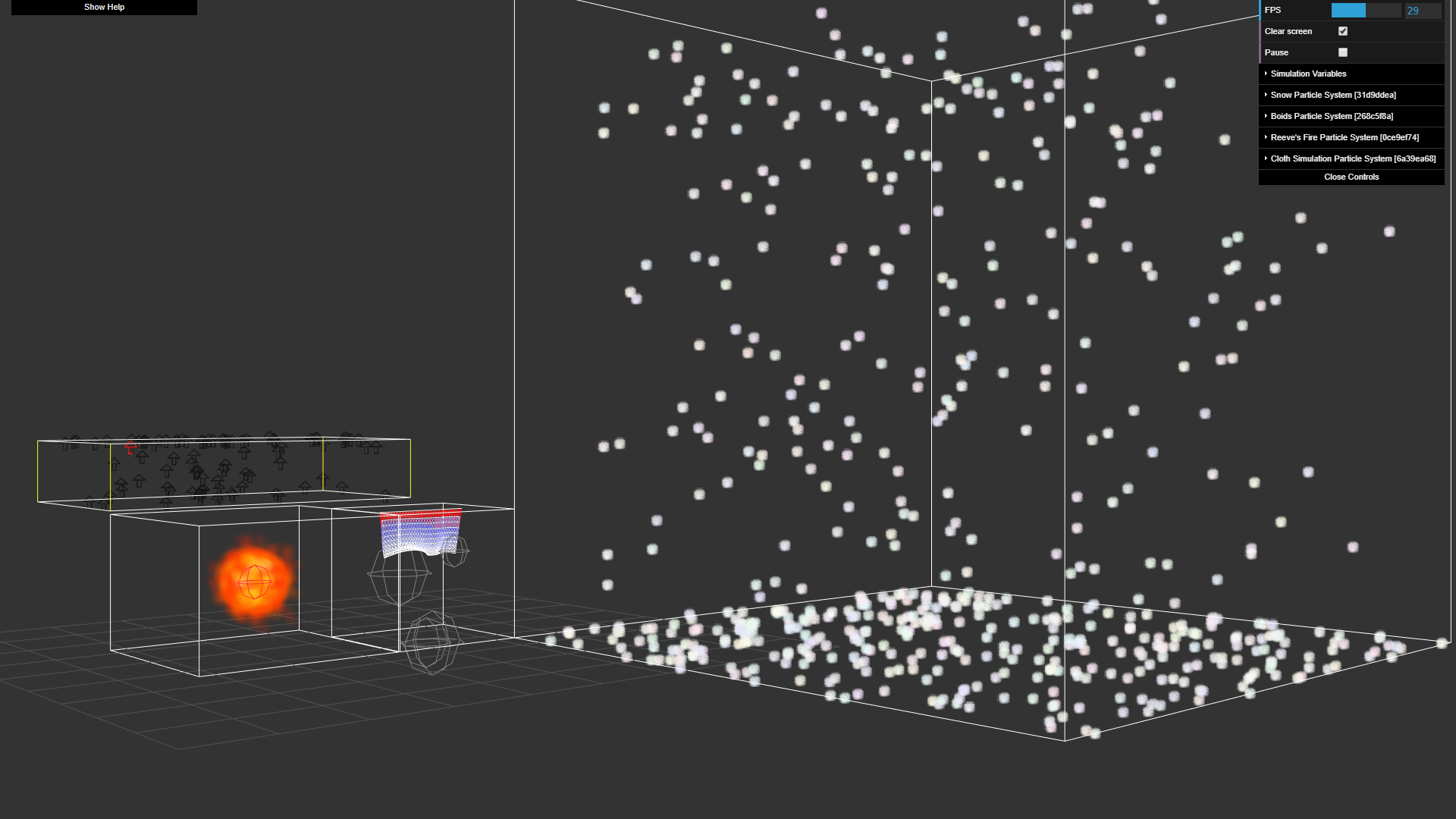


Figure 1: A broad view of all four particle systems.

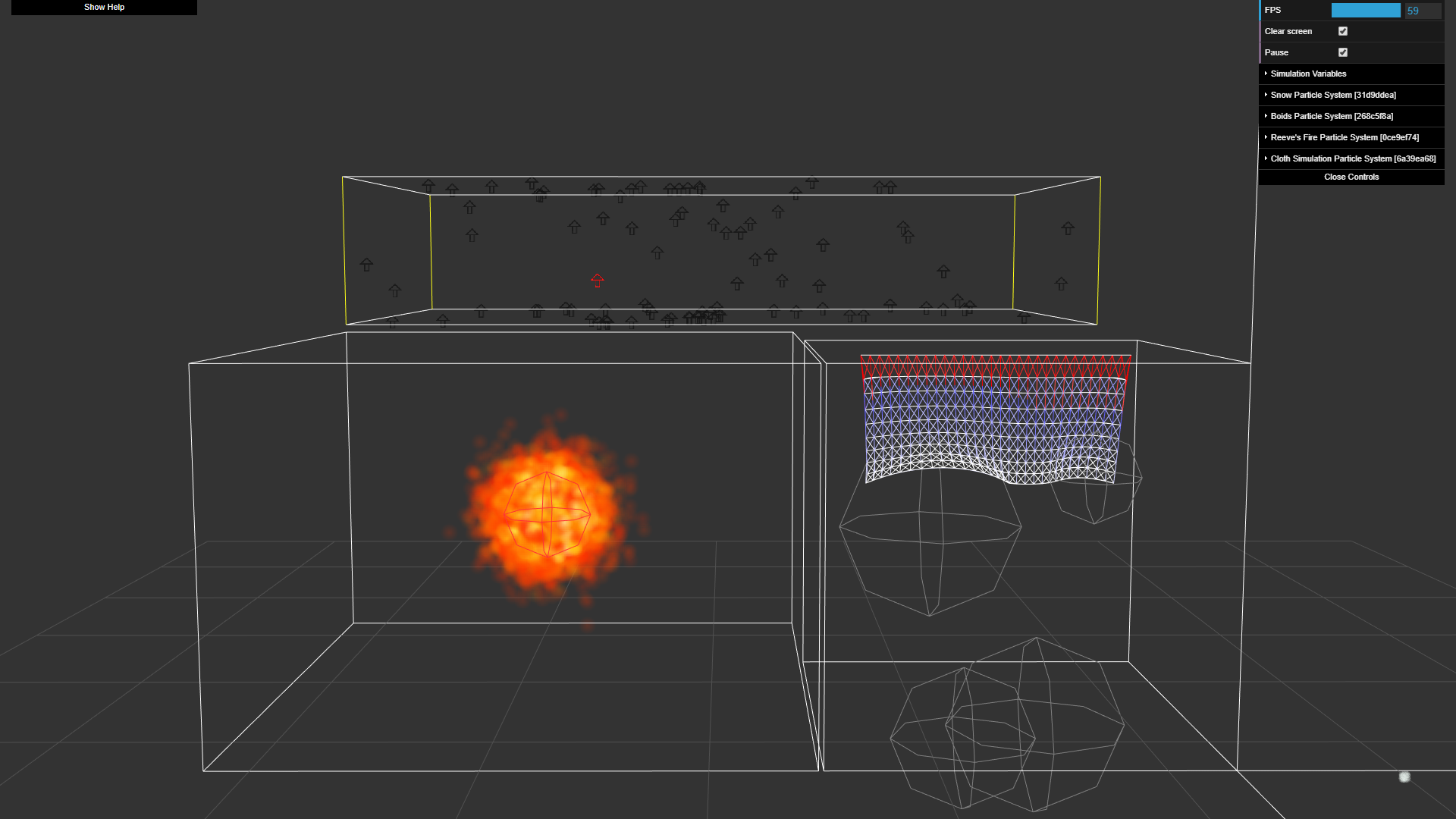


Figure 2: A side view of three particle systems. They are each contained in their own volumetric constraints, and boids can be seen avoiding the “predator.”

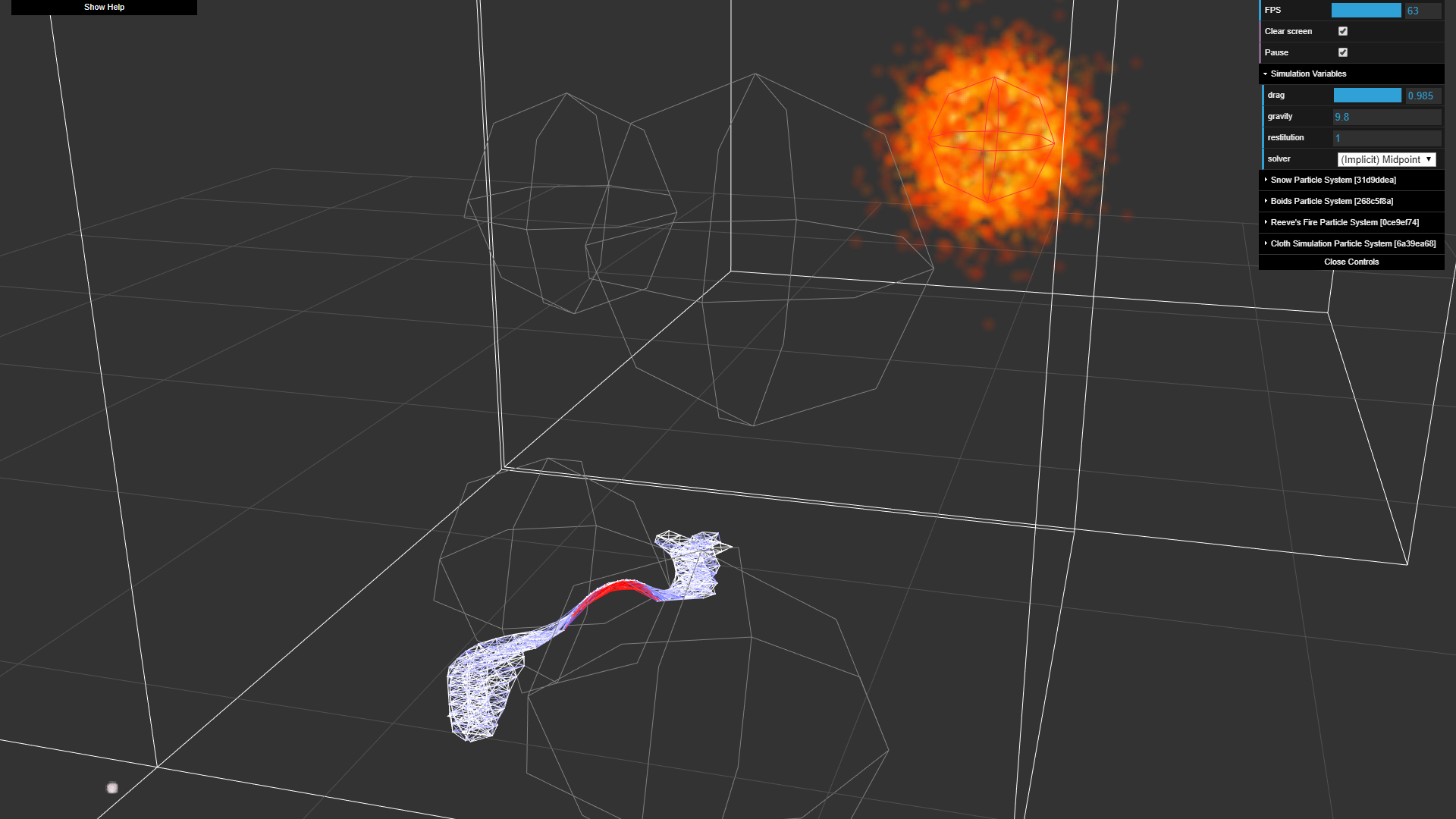


Figure 3: Cloth simulation settling on complex terrain. The springs are the most stretched out in the red central area. A more stable implicit solver gives good results.

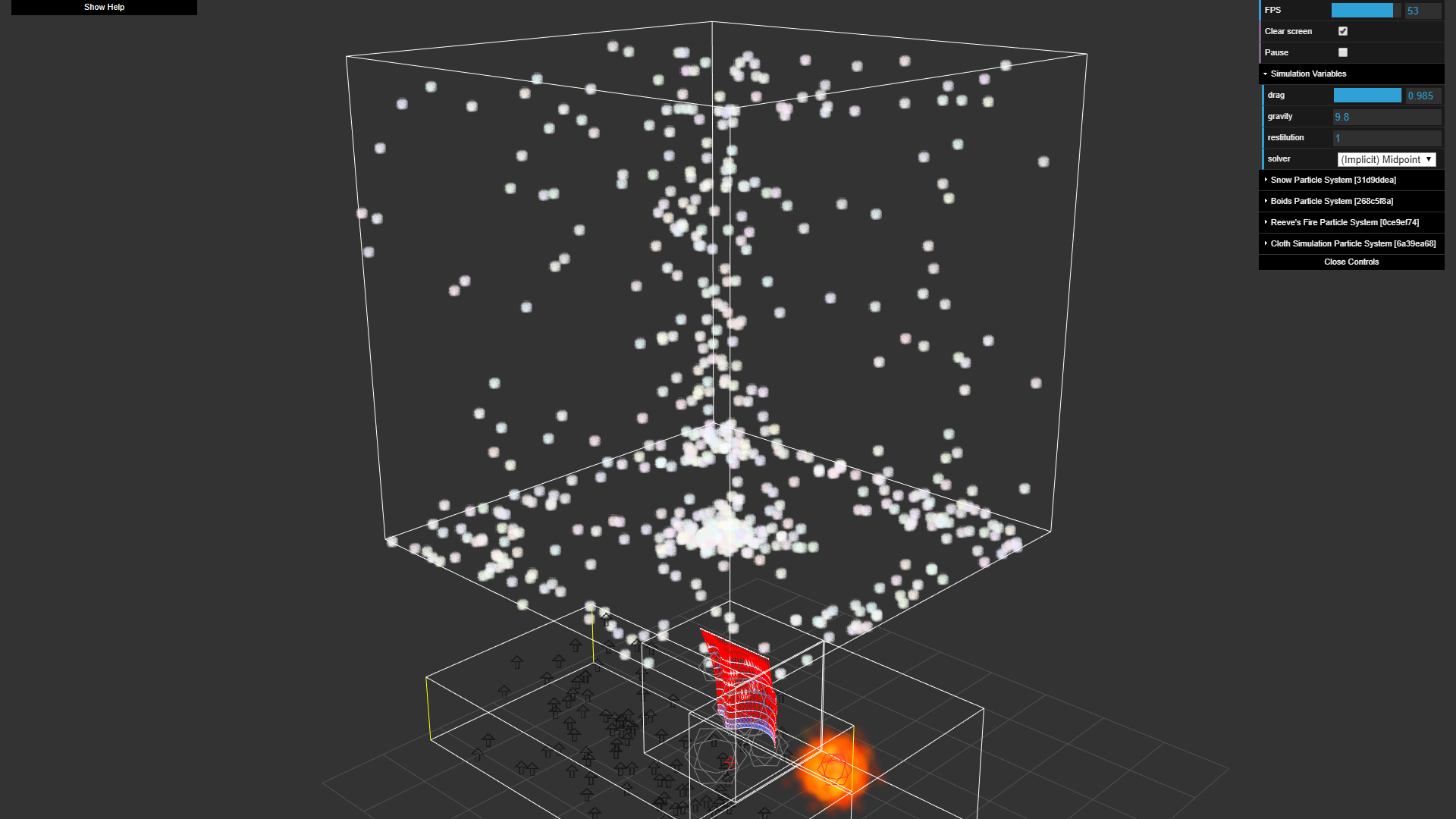


Figure 4: A vortex forming in the snow particle system. Snow particles are render using sprites with fuzzy edges.



Figure 5: Fewer constraints in the way, making for a cleaner appearance. Sprites are used to render boids and fire particles, while spring particles are rendered as only a single white pixel.

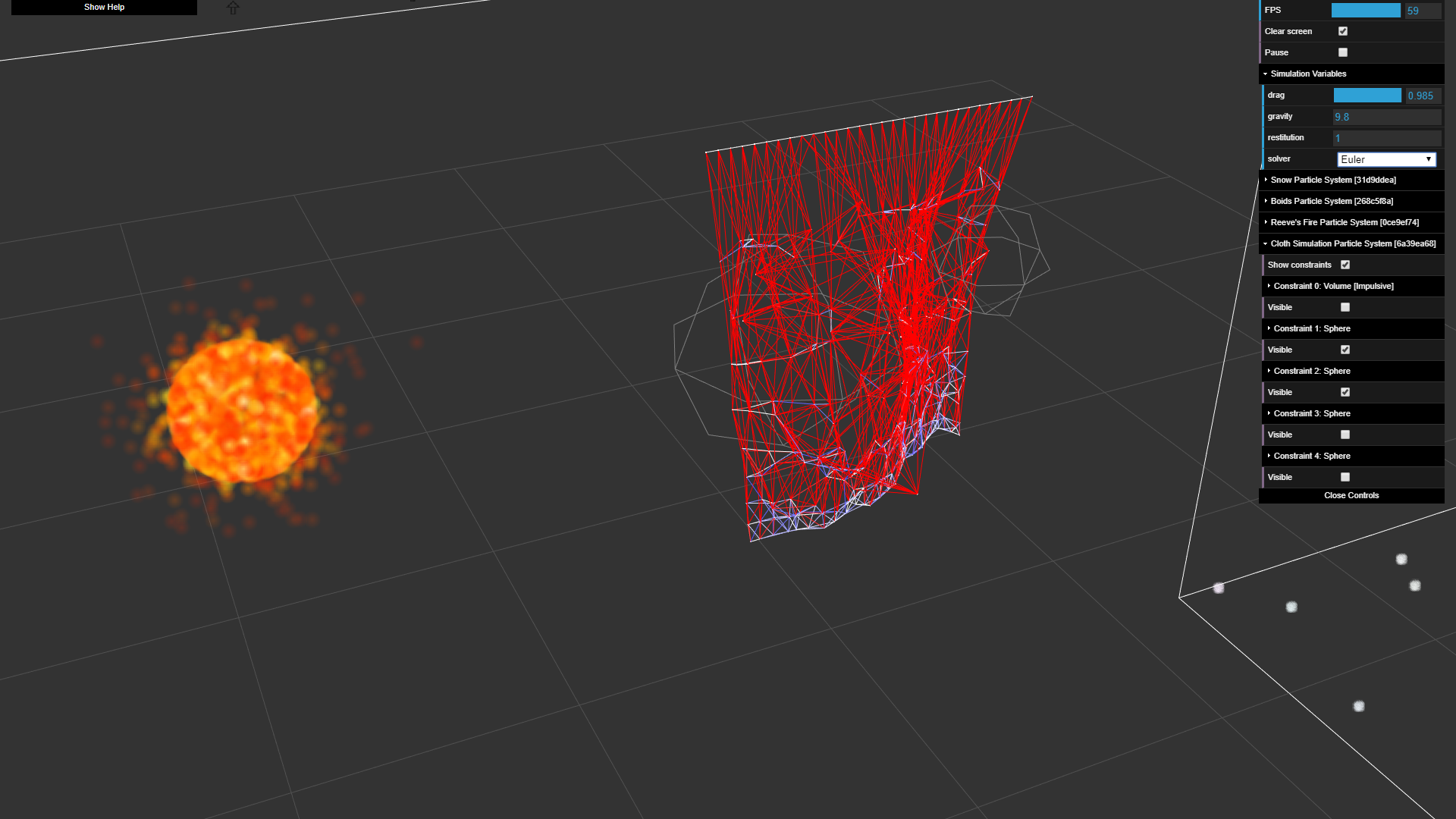


Figure 6: Instability in springs when explicit Euler integration is used.

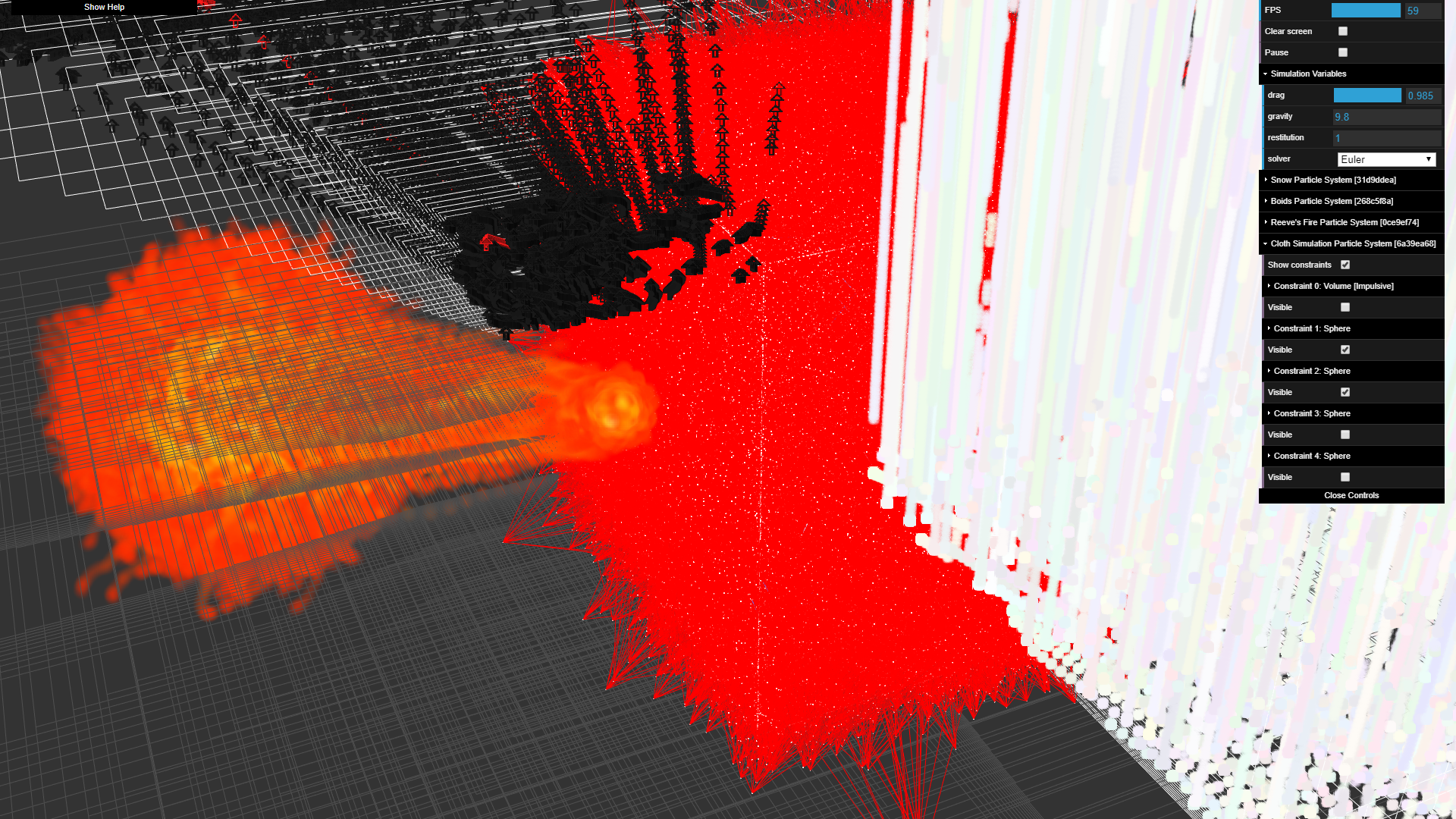


Figure 7: Just for fun.