Constraint fluids in Sprinkle

Dennis Gustafsson
Mediocre

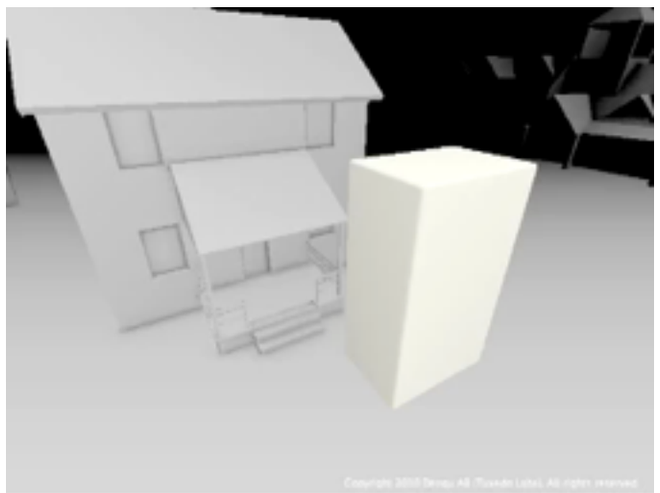
GAME DEVELOPERS CONFERENCE
SAN FRANCISCO, CA
MARCH 17–21, 2014
EXPO DATES: MARCH 19–21
2014

- Sprinkle. Sequel. Put out fires. Makeshift firetruck. Distant moon of Saturn.
- Fluid sim used at the core. Not only to put out fires -> move obstacles, flip switches, solve puzzles.
- Poll how many have seen or played.

Presentation outline

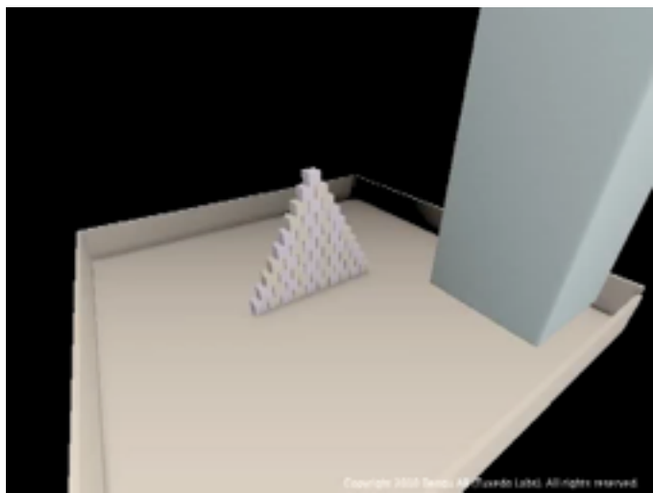
- Background and motivation
- Simulating fluid with constraints
- Implementation in Sprinkle
- Rendering and performance consideration

Experiments in 3D



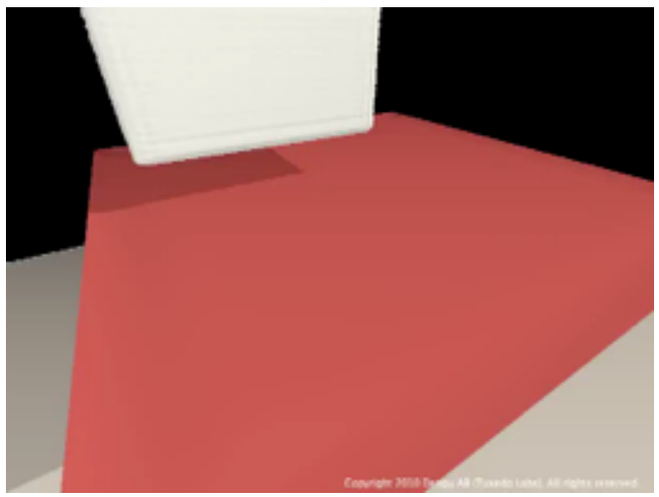
- Four years ago
- All types of physics, same unified solver. RB, cloth, rope, soft body and fluid
- Relatively incompressible, but hard to see. Old movie

Experiments in 3D



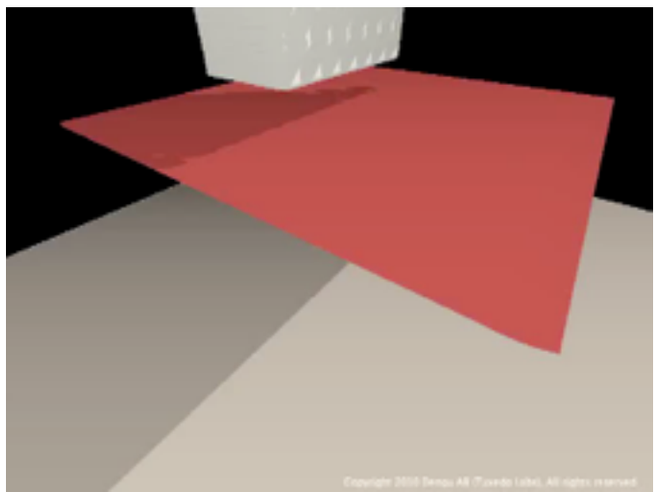
- See advatages of unified solver when interacting
- Separate may be faster
- Unified: stable interaction, no parameter tweaking

Experiments in 3D



- Fluid, cloth
- No more difficult if unified solver

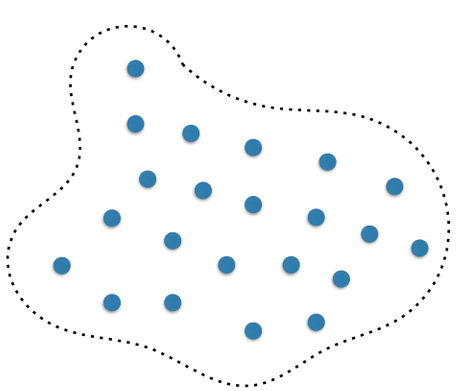
Experiments in 3D

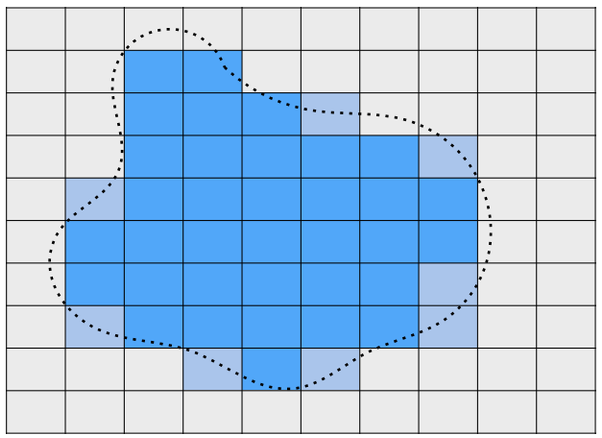


- Fluid, cloth, RB
- Not real-time

- 1st iPad
- HKJ
- Many iterations

Simulation paradigm

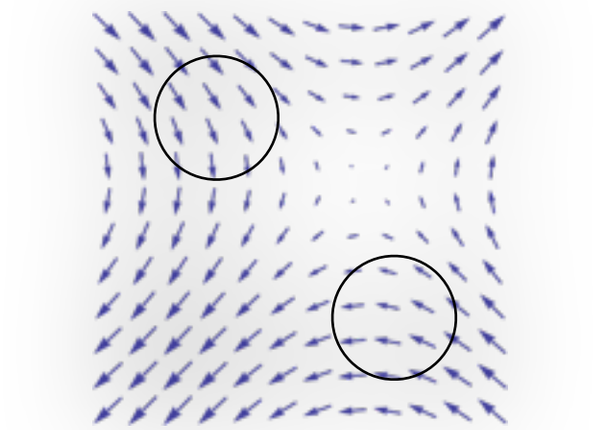




5 min

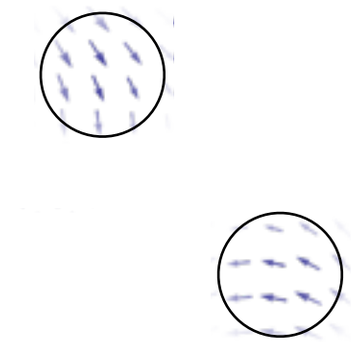
- Particle-based (SPH).
- Eulearian, or grid-based (Jos Stam, GDC03).
- Particle-based better for smaller simulations.

SPH particle ≠ droplet



- Smoothed Particle Hydrodynamics, astrophysics.
- Sample continuous field.
- Field implicitly defined by particles.
- Point-mass droplets not SPH

SPH particle ≠ droplet



- Windows into a continuous field.
- Difference not obvious, revisit later.

Traditional SPH solver recap

1. Find particle neighbors
2. Compute density at each particle
3. Compute and apply pairwise interaction forces
4. Integrate forces to new particle positions

•Traditional SPH vs constraint fluid, terminology.
•Steps, typical explicit method
•Good incompressibility -> iterative a few times dep on amount, gravity, etc

Rigid body solver recap

1. Find body neighbors
2. Setup velocity constraints
3. Sequential impulse solver: Apply impulses at contacts until all are separating*
4. Integrate velocities to new position and orientation

Steps

Conceptual differences

- SPH: Particle positions affect interaction forces. Forces are integrated.

- Rigid body: Velocity of other bodies affect impulses. Velocity is integrated.

- Similarities, differences.
- SPH: Position in, force out. RB: Velocity in, velocity out.
- Why this matters -> iteration.

Expriment

What would a rigid body simulator look like if implemented the same way as SPH?

- Col det. Penalty force based primarily on pen depth. Integration.
- Small time step, tweak parameters spec for each simulation.

Penalty method

- Springy behavior
- Rigid bodies are supposed to be **rigid**

- Liquids are **not** rigid
- …but liquids are **incompressible**

- Penalty same as spring
- Bad choice for rigid bodies
- Good for compressible fluid like smoke
- Liquids have similarities to rigid bodies
- Not rigid but incompressible

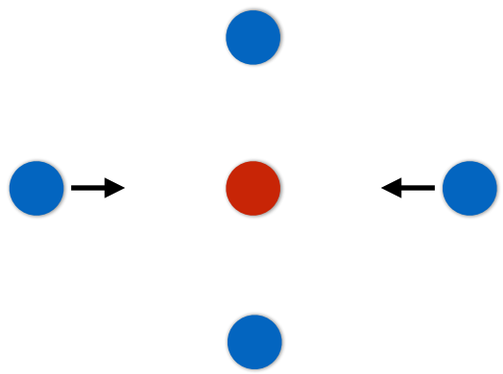
How can we model fluid motion as a velocity constraint?

- Motivation for fluid as velocity constraint.
- Not only RB interaction
- Faster (multiple iterations)
- Not springy
- Less parameter tweaking

Pair-wise interaction is not enough

15 min

- Pairwise constraint, much like frictionless rigid body.
- Momentum is not preserved. Internal motion dies out.

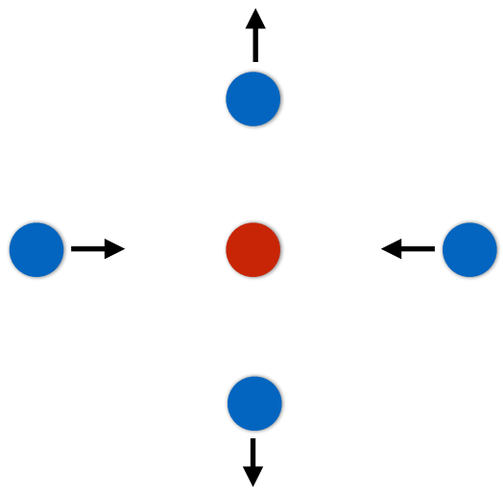


- Motion cancels out

A fluid particle is not a point mass.
It’s a discretization of a field.

- Recap SPH particle
- Finite amount of fluid, but not the same fluid
- Fluid can flow through the particle

Fluid can flow through the particle



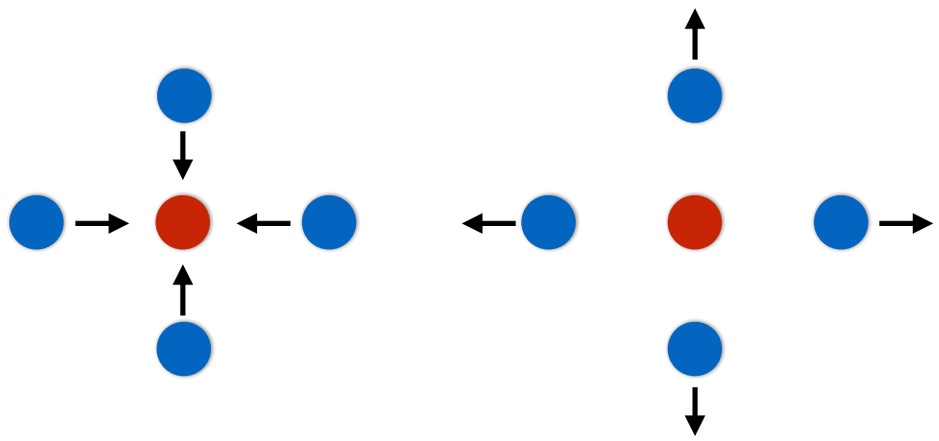
Incompressible flow

- Fluid can flow through a particle, but the density should remain constant.

$$\frac{D\rho}{Dt} = 0$$

- Density constant. Constrain first derivative, density change instead
- Compare this to RB, penetration vs contact velocity
- Whatever flows in also has to flow out

Motion of neighboring particles affect the change in density.



- Motion of neighbors affect density change
- Altering the motion of neighbors gives us a tool to control density change

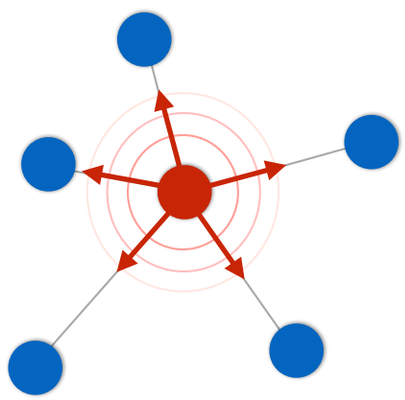
Constrain the motion of neighboring particles so that the net change in density is zero.

- The idea is to constrain…
- This constraint involves all neighbors -> not per pair, per particle.

The particle itself *is* the constraint

- When we satisfying a fluid constraint we make sure the change in density is zero.

Pressure as impulse



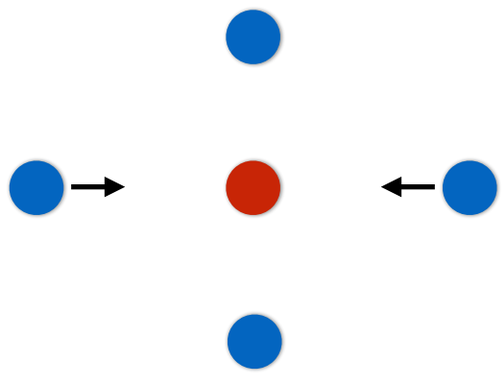
- RB: Equal, opposite impulses
- Pressure affects all neighbors in direction of neighbor
- Tangential motion unaffected -> frictionless RB
- Choose pressure so that density change is exactly zero.

Constraint formulation

Rigid body: Find the impulse magnitude, to apply to *both* bodies, so that the relative velocity at the contact point is zero.

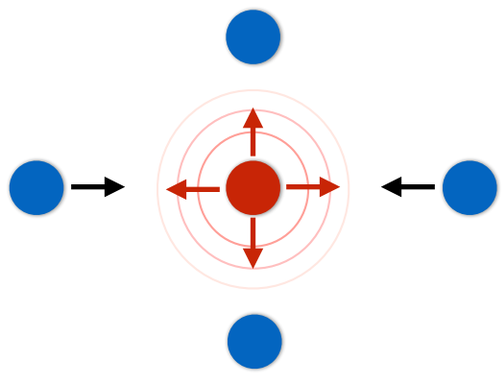
Fluid constraint: Find the pressure, to apply to *all neighboring* particles, so that the net change in density at the particle position is zero.

Example



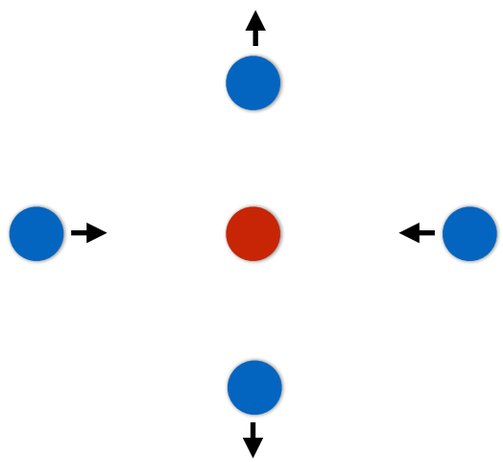
- Increasing density.
- Compute pressure needed to counteract density change.

Example



- Apply pressure impulse to all neighbors

Example

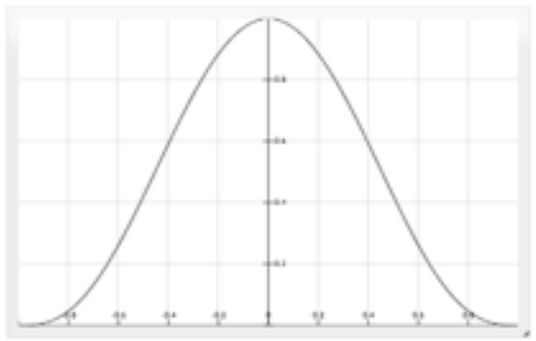


- Some of the horisontal motion has been converted into vertical motion.
- What flows in from the sides flows out at the top and bottom.
- Net density change is zero.

Example

- Reasonably incompressible
- Momentum preserved
- Wavefronts formed

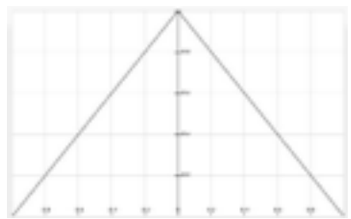
Use a smoothing kernel to avoid discontinuities.



$$(1 - (d/h)^2)^3$$

- Particles far away should not interact.
- Use kernel function to scale interaction.
- Zero = particles overlap, full interaction.
- One smoothing distance = zero interaction.
- Traditional SPH sensitive to kernel function. Not constraint fluid.

$$w = 1 - d$$



- Simplest possible kernel function

Iterative solvers aren’t perfect

Rigid body simulators compensate for geometric errors (inter-penetration)

The same can by done with a fluid constraint by compensating for deviations in density.

25 min

- Iterative solvers not perfect
- Rigid body solver compensate for penetration -> Baumgarte stabilization (velocity steering)
- Same can be done for fluid -> compensate for deviations in density

Constraint setup

```
for each particle p
        bias[p] = (restDensity-density)*baumgarte
        for each neighbor n
                d = distance/smoothingLength
                weight[n] = (1-d^2)^3
                A[p] += 2 * weight[n]^2
        next
next
```

- Velocity steering
- d – fraction of smoothing length
- Weight – kernel function
- Effective mass

Solver iteration

```
for each particle p
        for each neighbour n
                dv = dot(direction[n], vel[n]-vel[p])
                dpSum += weight[n]*dv
        next
        target = dpSum + bias[p]
        magnitude = max(0, target / A[p])
        for each neighbour n
                vel[n] += dir[n] * magnitude * weight[n];
                vel[p] -= dir[n] * magnitude * weight[n];
        next
end
```

measure density change

- Three steps
- Compute change in density
- Sum up projected relative velocity scaled by the weight

Solver iteration

```
for each particle p
        for each neighbour n
                dv = dot(direction[n], vel[n]-vel[p])
                dpSum += weight[n]*dv
        next
        target = dpSum + bias[p]
        magnitude = max(0, target / A[p])
        for each neighbour n
                vel[n] += dir[n] * magnitude * weight[n];
                vel[p] -= dir[n] * magnitude * weight[n];
        next
end
```

compute pressure impulse

- Target density change
- Compute magnitude
- Do not allow negative impulses

Solver iteration

```
for each particle p
        for each neighbour n
                dv = dot(direction[n], vel[n]-vel[p])
                dpSum += weight[n]*dv
        next
        target = dpSum + bias[p]
        magnitude = max(0, target / A[p])
        for each neighbour n
                vel[n] += dir[n] * magnitude * weight[n];
                vel[p] -= dir[n] * magnitude * weight[n];
        next
end
```

simplified!

- Should clamp accumulated impulse instead

Solver iteration

```
for each particle p
        for each neighbour n
                dv = dot(direction[n], vel[n]-vel[p])
                dpSum += weight[n]*dv
        next
        target = dpSum + bias[p]
        magnitude = max(0, target / A[p])
        for each neighbour n
                vel[n] += dir[n] * magnitude * weight[n];
                vel[p] -= dir[n] * magnitude * weight[n];
        next
end
```

apply pressure impulse

- Apply pressure impulse
- Newtons third law

1 iteration

- To achieve good incompressibility a few iterations are needed
- Same as RB solver with stacking
- Having problems carrying its own weight

2 iterations

- Momentum better preserved
- Maintains the volume better

4 iterations

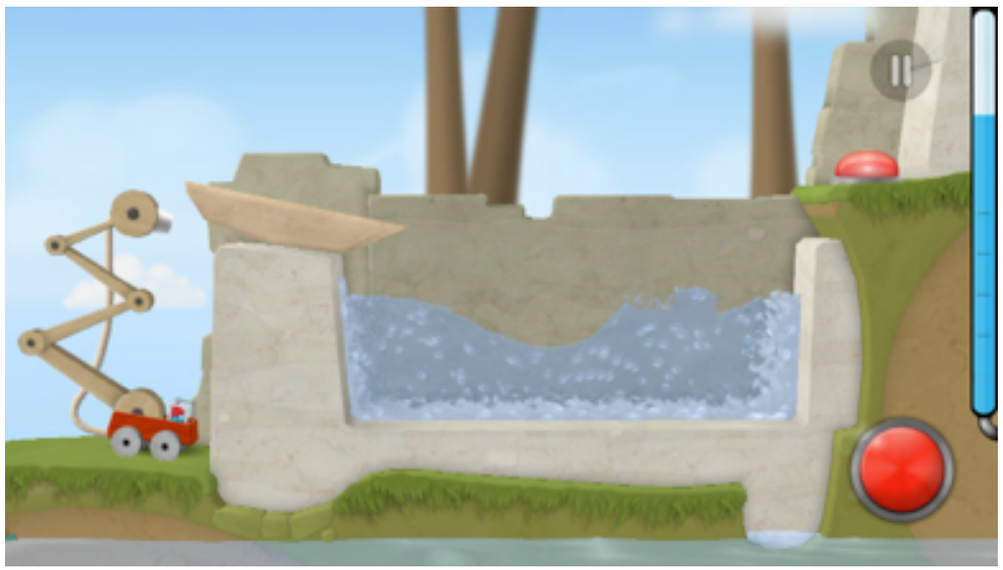
- Good incompressibility
- Momentum is conserved

- Wavefronts form
- Reflected on walls
- Happens automatically

Sub-optimal implementation in Sprinkle

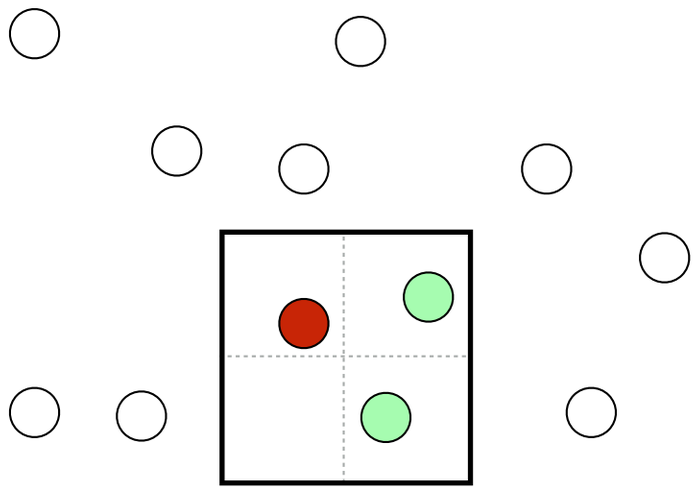
34 min

- Implementation in Sprinkle
- Box2D. Separate solvers
- Enough for this game



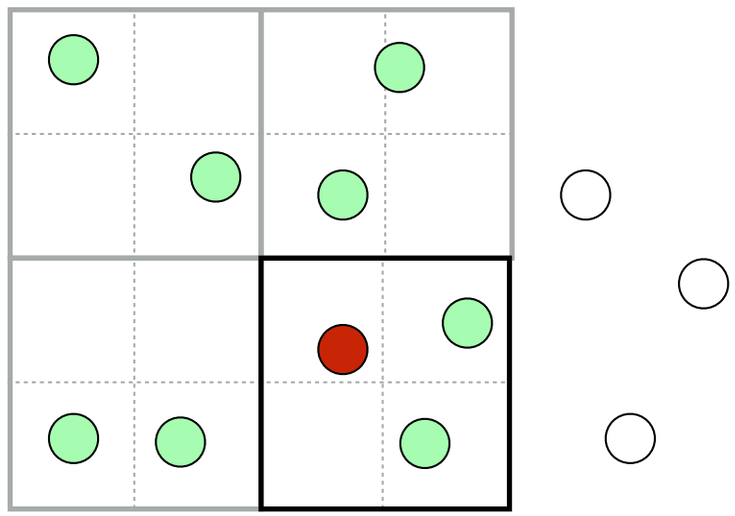
- Note the instability

Spatial binning with quadrants



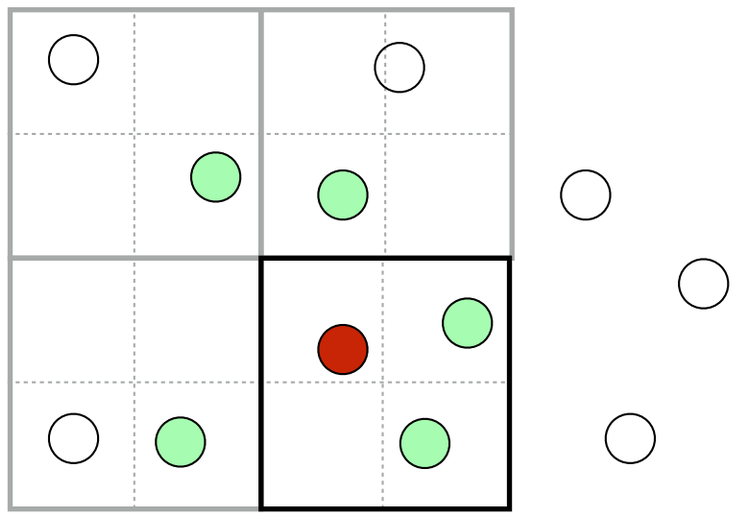
- Neighbor finding
- Spatial binning, fixed grid size
- Keep track of quadrant

Spatial binning with quadrants



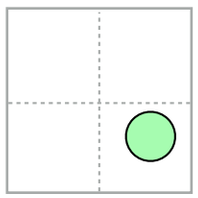
- Particle in top left quadrant can only interact with particles above or to the left.

Spatial binning with quadrants



- Distance check quadrants facing the original particle
- Instead of all particles in eight cells, check half of particles in three cells

Local quantized cell coordinates



(89, -50)

- Compact 8-bit representation
- Determine quadrant by analyzing sign

- During neighbor finding, use local quantized coordinate
- Sign gives quadrant
- Compact 2-byte representation per particle. Cache friendly.

Collision detection

- Reuse binning grid cells
- Box2D broad phase to collect shapes
- Dual representation for convex shapes - bounding planes

- Collect rigid bodies by querying the Box2D broad phase with grid cells
- Separate representation for fluid collision
- Plane equations instead of vertices

Solving collisions

- Rigid body interaction as body/particle constraint
- Solve contacts **after** fluid constraints

- Velocity constraint
- Like a small, rotationless rigid body without friction
- Solve collisions after fluid constraints

Memory layout

```
struct Particle
{
  vec2 position;
  Neighbor neighbors[MAX_NEIGHBORS];
  int neighborCount;
  Precomputed stuff;
};
```

```
struct Neighbor
{
  int index;
  float weight;
  vec2 direction;
};
```

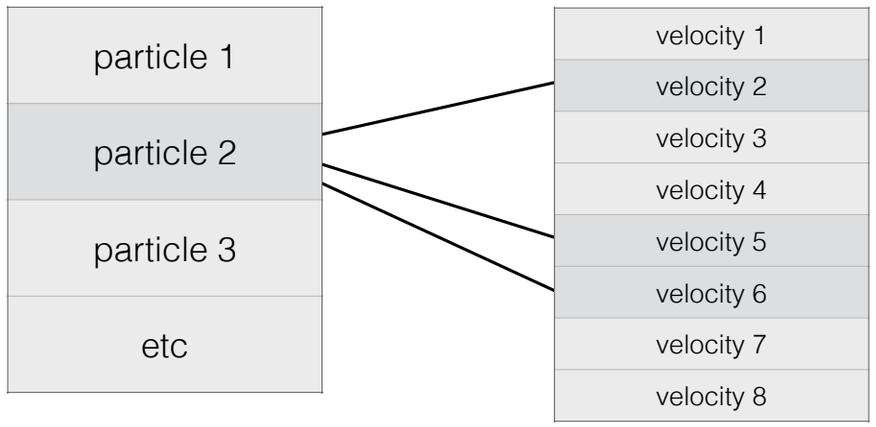
- Precomputed neighbor information stored in each particle
- Some information duplicated, but linear traversal

Memory layout

```
Particle particles[MAX_PARTICLES];
vec2 velocities[MAX_PARTICLES];
```

- Velocities kept in a separate array

Memory layout



- Particle array traversed linearly
- Neighbor velocities gathered, scattered

Rendering

- Limited by hardware at the time
- No marching squares. No multipass
- Traditional point splatting with a few tricks

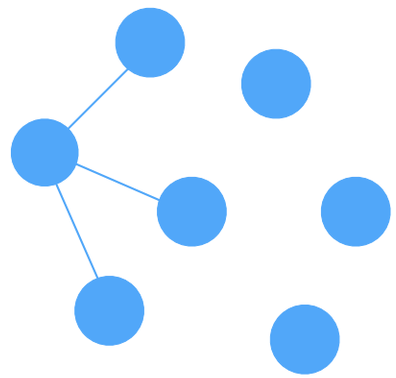
We have a lot more information than just particle position!

Density — particle size

Pressure gradient — particle orientation

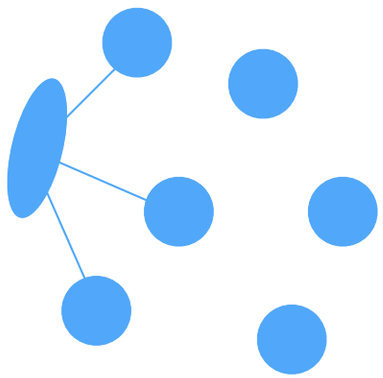
- Neighbor finding and constraint setup gives more information then just position

Particle orientation



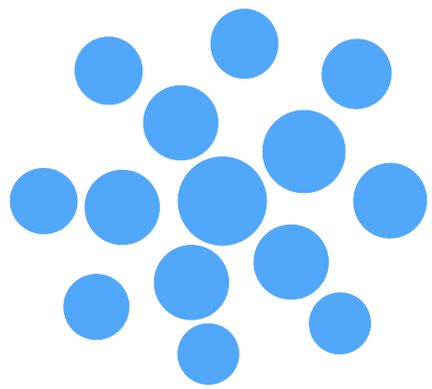
- Analyze average neighbor distance vector.
- If edge particles —> stretch and rotate in direction of edge tangent.

Particle orientation



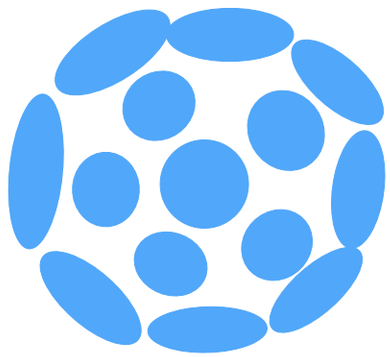
- Smoother looking surface. Allows for bigger particles.
- Render upward facing particles in brighter color to emulate highlight from sun.

Particle orientation



Bigger example.

Particle orientation



- Neighbor configuration changes often
- Temporal coherence. Allow small rotations
- Relaxation step for smoother edges

Splashes

- No fluid simulation
- Ballistic motion
- Full collision detection
- Removed upon collision

- 600 simulated (1st gen), 800 in islands (2nd gen)
- Splashes for more visual detail. Simple ballistic trajectories. Removed on collision
- Emit splashes at low-density fluid particles moving upwards
- Smaller and brighter color

Particles

- Raw fluid particles

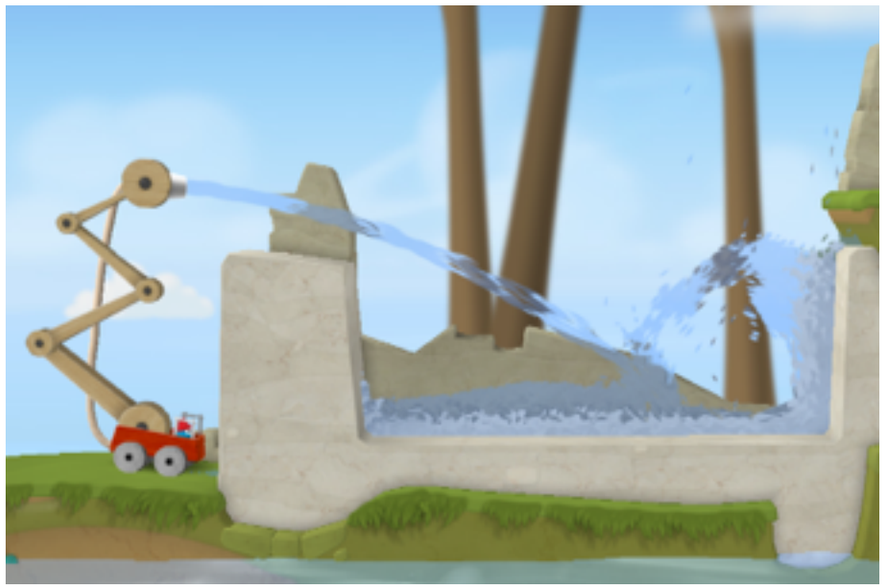
Particles Resize

- Resized by density
- Gives stray particle less attention

Particles
Resize
Stretch

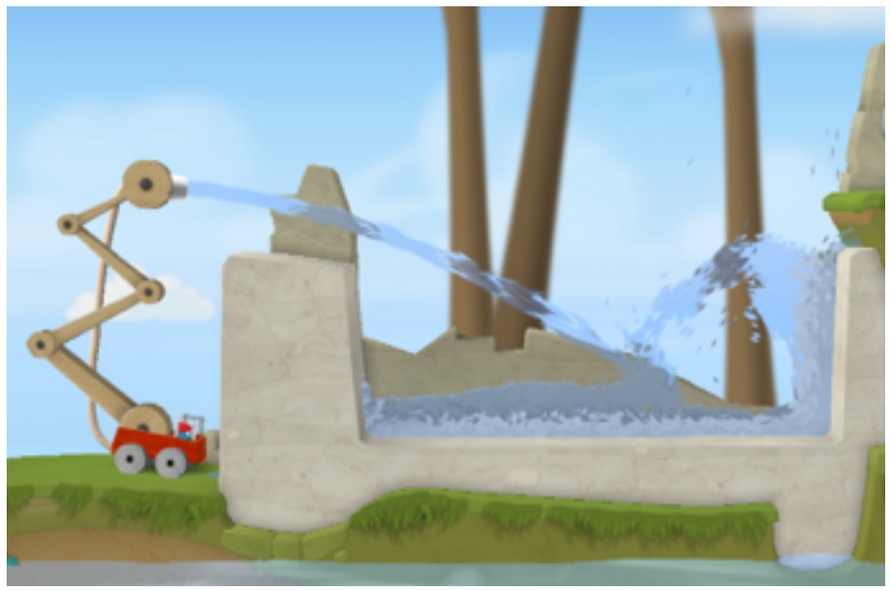
- Velocity and edge stretch
- Makes it less bubbly

Particles
Resize
Stretch
Refraction



- Refraction
- Distort texture coordinates

Particles
Resize
Stretch
Refraction
Edges

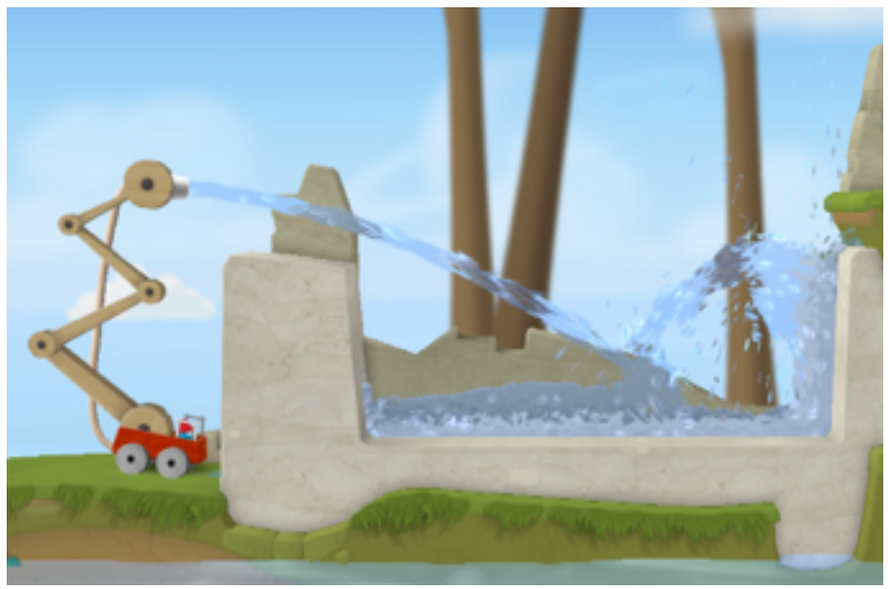


- Edge particles
- Fake highlight

Particles
Resize
Stretch
Refraction
Edges
Splashes

- Splash particles

Particles
Resize
Stretch
Refraction
Edges
Splashes
Bubbles



- Some high density particles rendered as bubbles
- Reveal internal motion.

Particles
Resize
Stretch
Refraction
Edges
Splashes
Bubbles

Thank you

dennis@mediocre.se
http://tuxedolabs.blogspot.com

- Questions