Particle Simulator: arte in movimento

Lorenzo Piarulli 1884766

Per ogni simulatore descritto di seguito, sono stati prodotti dei video dimostrativi riportati nella cartella "CreativePreviews". Inoltre, in questo paper sono riportati i frames più significativi. Tutti i frames prodotti sono contenuti in "CreativeFrames".

Per ottimizzare l'esecuzione di alcuni simulatori è stata modificata una funzione fornita dal docente. Tale modifica riduce il numero di **positions** in una **particle shape**.

```
vector<vec3f> p;
if (params.init_type != initializer_type::classic &&
    params.solver != particle_solver_type::simulate_gas &&
    params.solver != particle_solver_type::simulate_oscillation) {
    rng_state rng = make_rng(params.seed);
    for (auto pos : ioshape.positions) {
        if (rand1f(rng) < 0.2) {
            p.push_back(pos);
        }
    }
} else {
    p = ioshape.positions;
}
add_particles(ptscene, iid++, ioshape.points, p, ioshape.radius, 1, 1);
```

Tied Neighbours Materials

Difficile

Prima di iniziare a descrivere gli algoritmi di simulazione sviluppati, viene presentato nel primo paragrafo un algoritmo inizializzatore. Quest'ultimo permette, utilizzando un simulatore **Position Based**, di ripensare la gestione delle **springs** in una formula più generale che permette la realizzazione di diversi materiali.

L'algoritmo genera per ogni particella, un insieme di particelle limitrofe calcolate in base ad un **range** assegnato. Gli insiemi generati prendono il nome di **Neighbours Set**, e vengono sfruttati per la costruzione delle **springs** dell'oggetto in questione.

Per facilitare la costruzione dell'inizializzatore, viene utilizzata una struttura dati per la rappresentazione dei **Neighbours Set**.

```
struct particle_neighbours {

  vec3f         particle;          // main particle
  int           index;             // main particle index
  vector<vec3f> neighbours;        // neighbouring particles
  vector<int>   neighIdxs;         // neighbouring particles indexes
```

```
};
```

Di seguito è presentato l'agoritmo di inizializzazione descritto.

```
void init_neighbours_simulation(
    yocto::particle_scene& scene, yocto::particle_params const& params) {

  reset_global_variables();

  auto      sid = 0;
  rng_state rng = make_rng(params.seed);

  for (auto& shape : scene.shapes) {
    shape.emit_rng = make_rng(params.seed, (sid++) * 2 + 1);
    shape.invmass  = shape.initial_invmass;
```

La porzione di codice seguente è necessaria per gli algoritmi **simulate_gas** e **simulate_tornado**, ma non di interesse per l'analisi di questo paragrafo. Gli algoritmi **simulate_tornado** e **simulate_gas** necessitano di variare randomicamente la massa di ogni particella. Inoltre per **simulate_gas** viene generato un vettore di temperatura, configurata in precedenza nei parametri, utile per la rappresentazione del calore di ogni particella.

```
    if (params.init_type == initializer_type::tornado) {
      for (int k = 0; k < shape.invmass.size(); k++) {
        shape.invmass[k] = rand1f(rng);
        if (params.solver == particle_solver_type::simulate_gas) {
          shape.invmass[k] /= 1000;
          temperatures.push_back(vec3f{0, params.temperature, 0});
        }
      }
      shape.emit_rngscale /= 5;
    }


    shape.normals      = shape.initial_normals;
    shape.positions    = shape.initial_positions;
    shape.radius       = shape.initial_radius;
    shape.velocities   = shape.initial_velocities;
    shape.bounce       = params.bouciness;
    shape.has_collided = false;
    shape.spring_coeff = 0.5;

    // Only for tornado simulation
    if (params.init_type == initializer_type::tornado) {
      shape.spring_coeff = 0;
```

```
    }

    shape.forces.clear();
    for (int i = 0; i < shape.positions.size(); i++)
      shape.forces.push_back(zero3f);

    for (int i = 0; i < shape.positions.size(); i++) {
      shape.positions[i] += rand3f(rng) / 20;
    }

    for (auto& velocity : shape.velocities) {
      velocity += sample_sphere(rand2f(shape.emit_rng)) *
shape.emit_rngscale *
                  rand1f(shape.emit_rng);
    }
```

Di seguito sono definiti i parametri fulcro di tutto l'algoritmo:

- Con **range** è definita la massima distanza nella quale pescare le particelle vicine.
- Con **rest_offset** è definito l'offset applicato al parametro **rest** di ogni **spring** utilizzata.

Modificando tali valori si ottenengono comportamenti estremamente diversi. Un **range** poco ampio permette di generare materiali poco densi, un **rest_offset** molto ampio dà la possibilità di modellare materiali più elastici.

```
    float range = 0.01;  // [0.001 , 0.01]

    float rest_offset = 0;  // [0 , 0.1]
```

L'algoritmo si conclude con la costruzione dei **Neighbours Set** e con la generazione delle **springs** di ogni **Neighbours Set**.

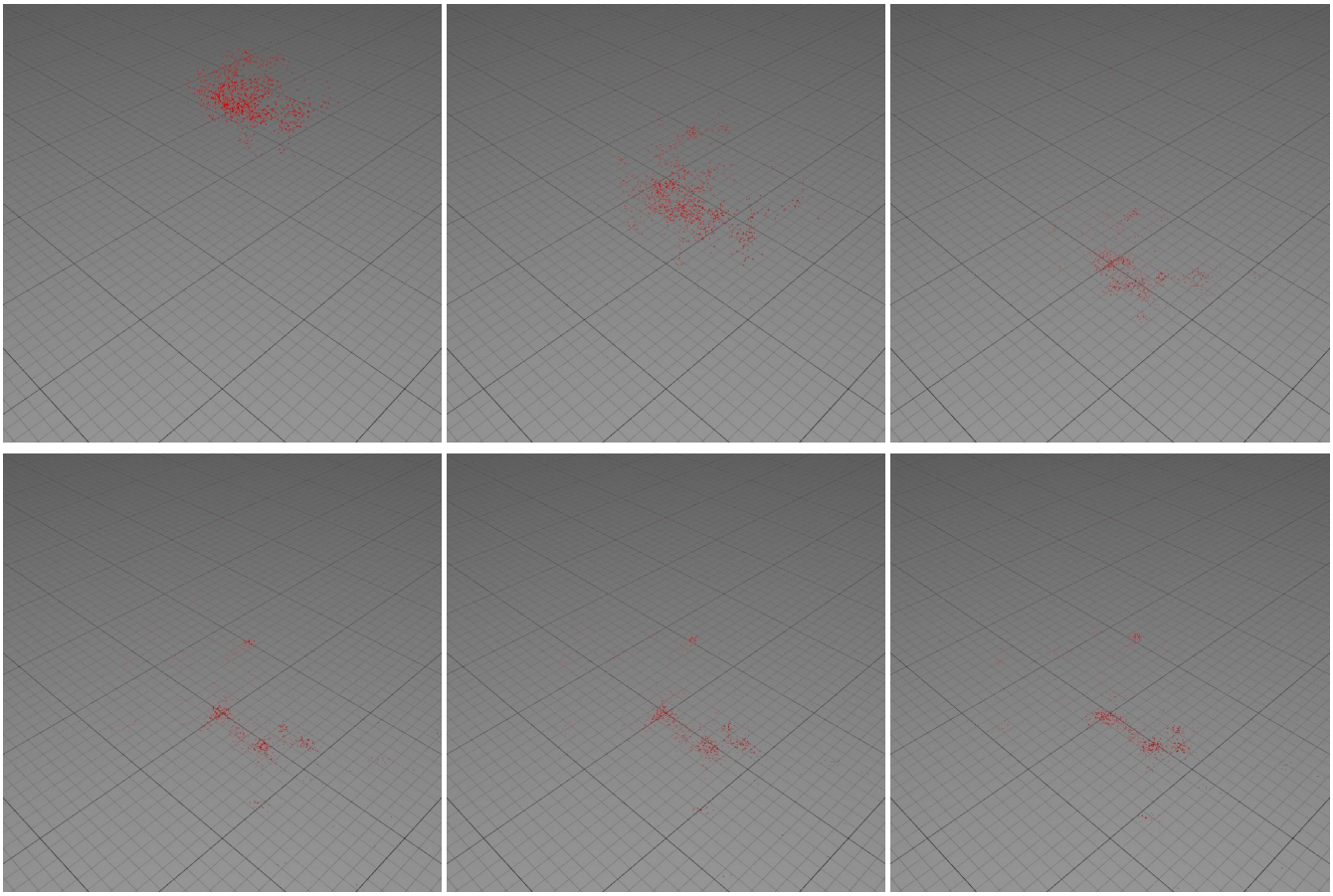
```
    for (int i = 0; i < shape.positions.size(); i++) {
      particle_neighbours particle;
      particle.particle = shape.positions[i];
      particle.index    = i;
      scene.groups.push_back(particle);
      for (int j = 0; j < shape.positions.size(); j++) {
        if (i != j &&
            distance(shape.positions[i], shape.positions[j]) < range) {
          scene.groups[i].neighIdxs.push_back(j);
          scene.groups[i].neighbours.push_back(shape.positions[j]);
        }
      }
    }
```

```
    shape.springs.clear();

    if (shape.spring_coeff > 0) {
      for (auto vert0 = 0; vert0 < shape.positions.size(); vert0++) {
        for (auto vert1 : scene.groups[vert0].neighIdxs) {
          particle_spring spring = {vert0, vert1,
              distance(shape.positions[vert0], shape.positions[vert1]) +
                  rest_offset,
              shape.spring_coeff};
          shape.springs.push_back(spring);
        }
      }
    }
  }

  . . .

  //Next istructions are init_simulation-like
}
```

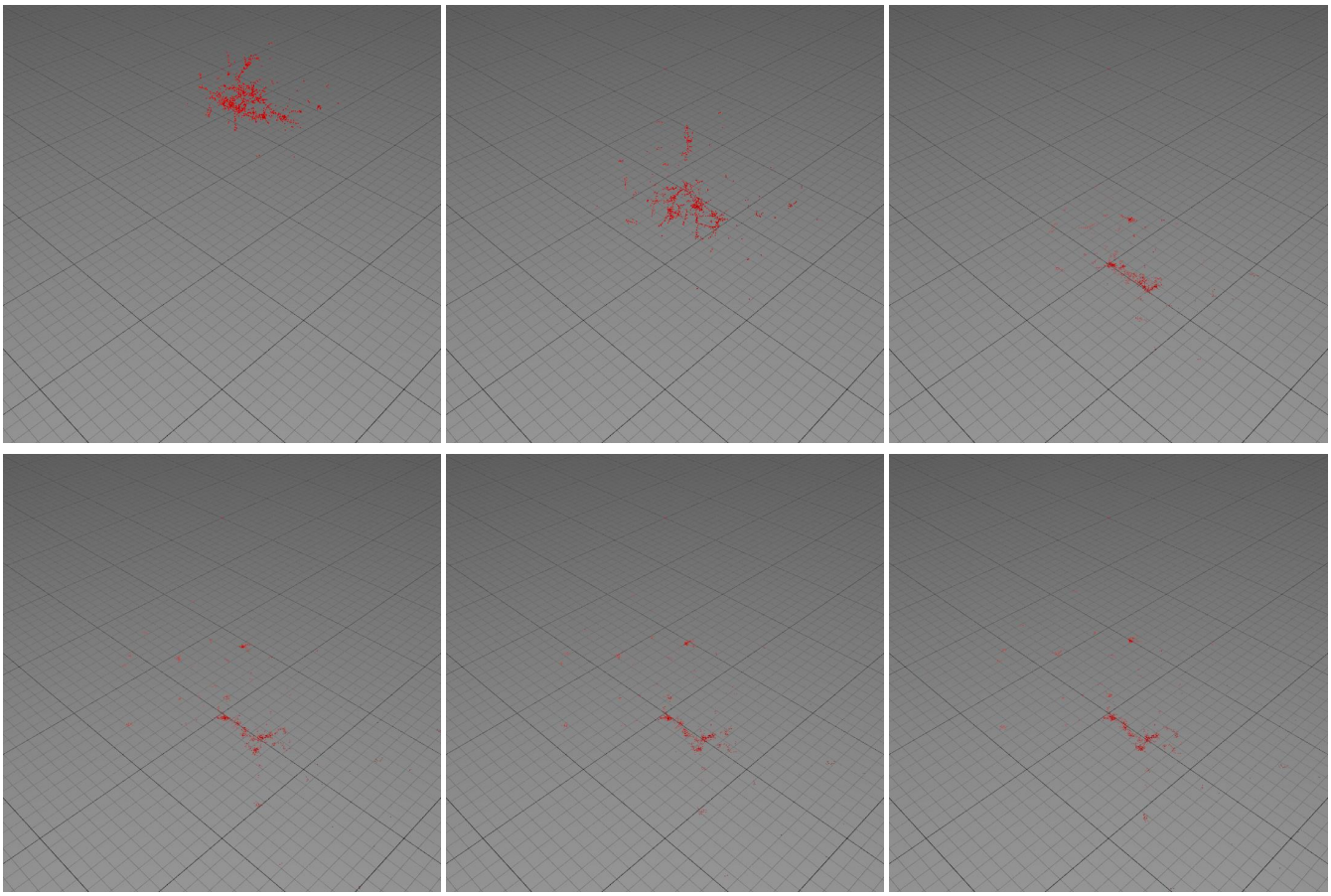
L'inizializzatore descritto in precedenza, come si è potuto riscontrare nel codice, viene sfruttato anche da altri simulatori che verranno analizzati nei prossimi paragrafi.

Di seguito i materiali e le animazioni ottenute:

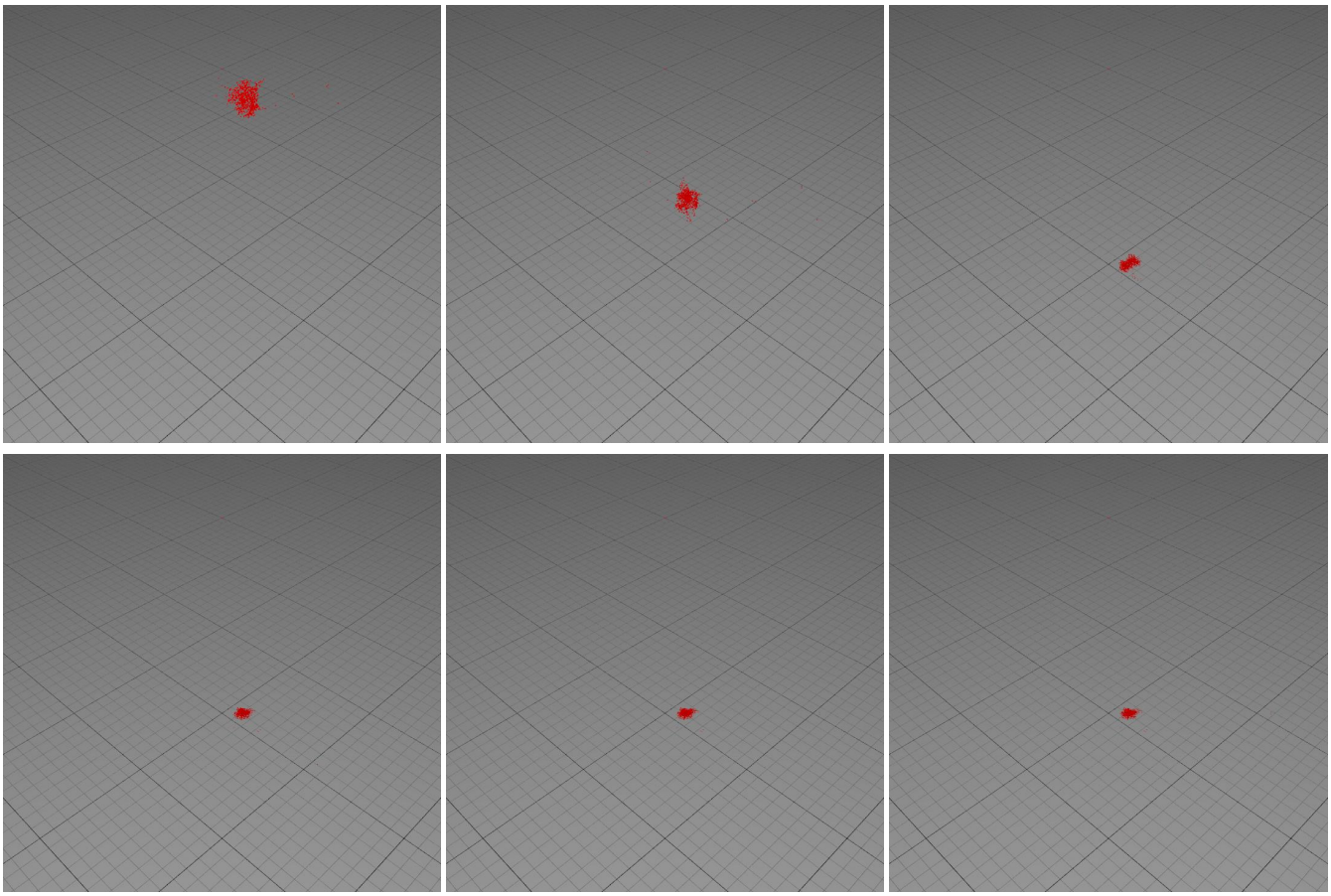
Water material: range = 0.005, rest_offset = 0.008



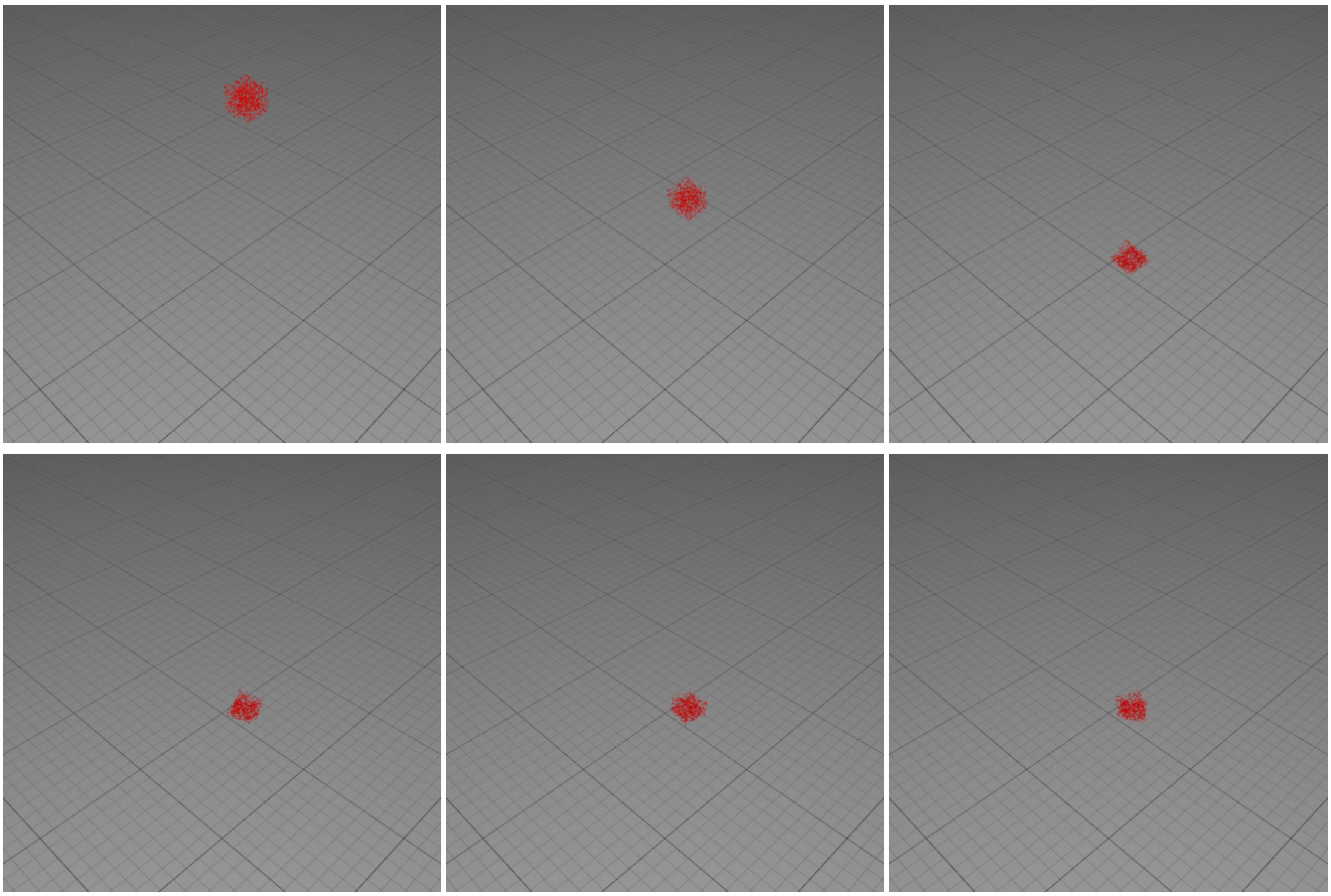
Paint-like material: range = 0.005, rest_offset = 0



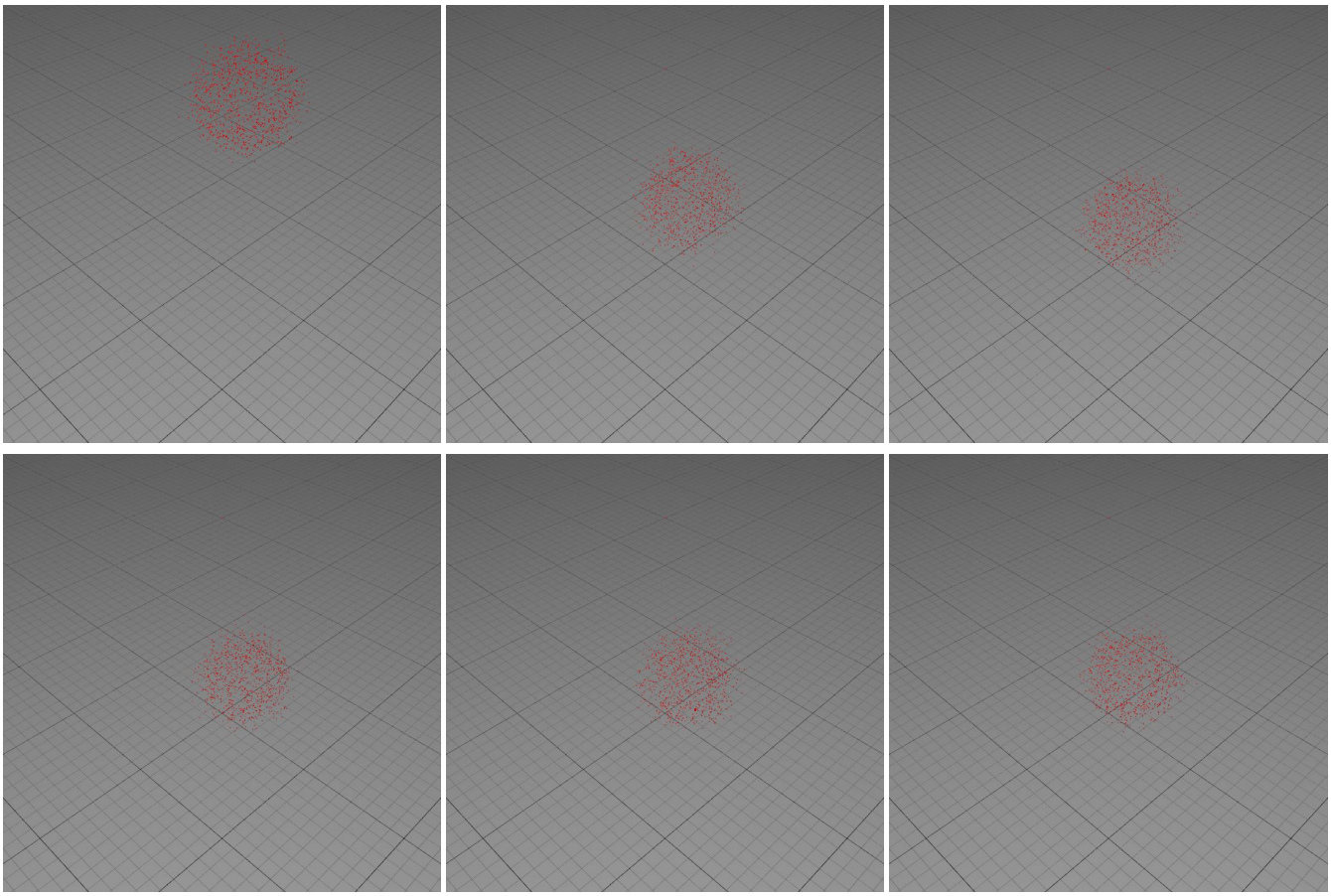
Slime material: range = 0.006, rest_offset = 0



Semi-rigid material: range = 0.01, rest_offset = 0



Softbody material: range = 0.01, rest_offset = 0.09



Floating Particles

In questo paragrafo vengono affrontati due algoritmi che, seppur con modalità e risultati differenti, generano delle particelle ultra leggere governate da agenti esterni.

Gas Behaviour Simulator

Medio

Il primo algoritmo di questa sezione simula il comportamento di particelle gassose, portate ad una temperatura definita. Il calore proprio di ogni particella permette a quest'ultima di elevarsi fino al suo raffraddamento, maggiore è la temperatura maggiore è la velocità e la durata dell'innalzamento.

Oltre al parametro della temperatura, sono necessari altri due vettori: **pressure_noise** e **air_resistance**. Il primo influisce sulla pressione subita dalle particelle nel loro atterraggio, al fine di spingerle a muoversi parallelamente al terreno. Il secondo permette di simulare la discesa delle particelle, rallentata dall'azione dell'aria che bilancia il loro peso.

```
vec3f pressure_noise = {1, 0, 1};
vec3f air_resistance = {0, 0.3, 0};

void simulate_gas(particle_scene& scene, const particle_params& params) {
  for (auto& shape : scene.shapes) {
    shape.old_positions = shape.positions;
  }

  rng_state rng = make_rng(params.seed);

  for (auto& shape : scene.shapes) {
    for (int k = 0; k < shape.positions.size(); k++) {
      if (!shape.invmass[k]) continue;
```

Calcolo della forza peso di ogni particella. *Invmass* rappresenta l'inverso della massa, per semplicità viene considerata come la massa effettiva.

```
      vec3f weightforce = {0, -params.gravity * shape.invmass[k], 0};
```

Applicazione dell'azione della forza peso e del calore.

```
      shape.velocities[k] += (weightforce) * params.deltat;
      shape.velocities[k] += temperatures[k] * params.deltat;
```

Applicazione del parametro **pressure_noise** per le particelle vicine al terreno.

```
      if (shape.positions[k].y < 0.15) {
        shape.velocities[k] +=
            (pressure_noise * normalize(shape.positions[k]) * rand1f(rng))
*
            params.deltat;
      }
```

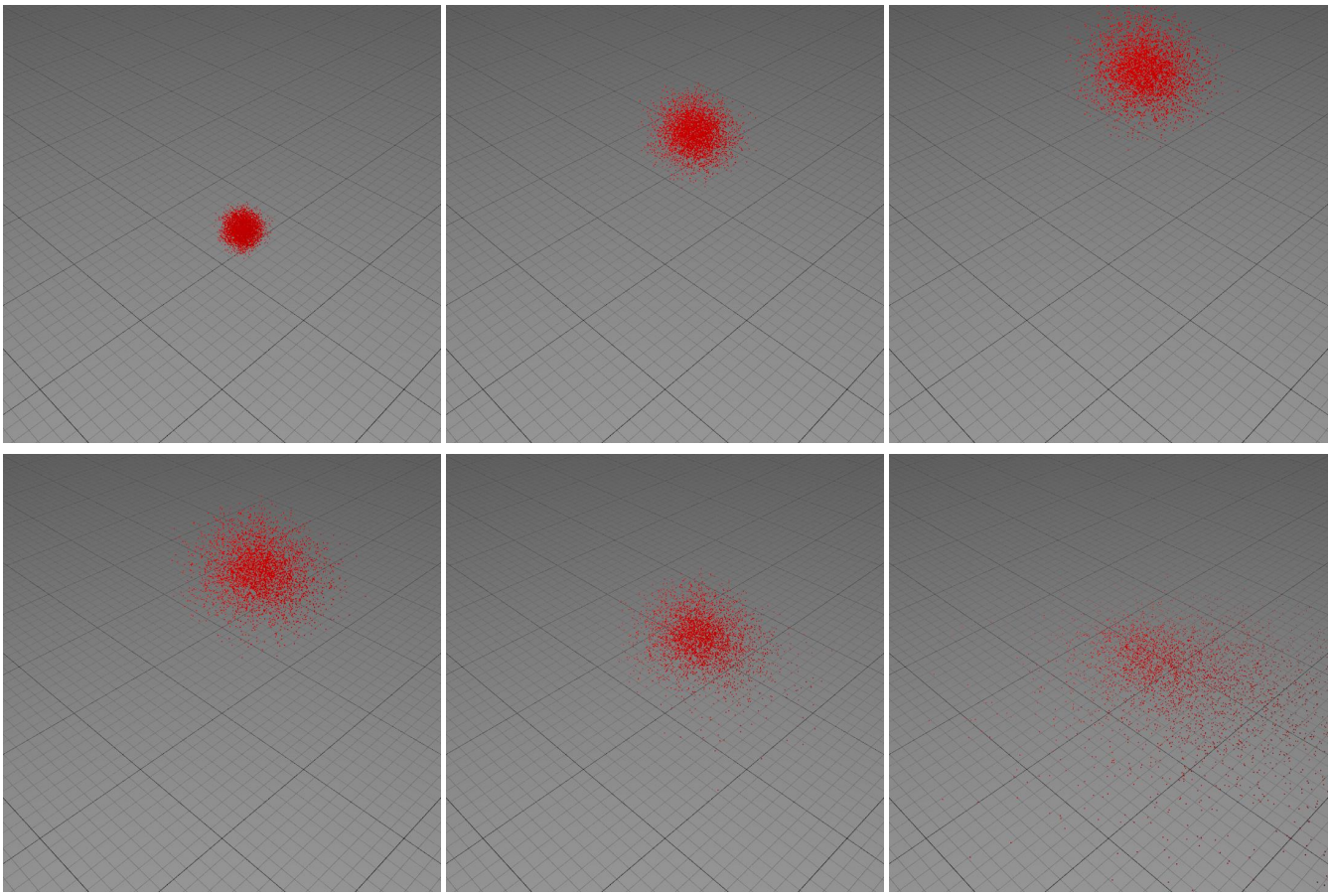
Applicazione del parametro **air_resistance** alle particelle discendenti.

```
      if (shape.velocities[k].y < 0) {
        shape.velocities[k] += air_resistance * params.deltat;
      }

      shape.positions[k] += shape.velocities[k] * params.deltat;
```

Decrescita della temperatura di ogni particella.

```
      if (temperatures[k].y > 0) {
        temperatures[k].y -= 0.4;
      }
    }
  }

  . . .

  // Next istructions are pbd-like
```

Di seguito sono riportati i risultati ottenuti con un valore di **temperature** fissato a *10*.



Paranormal Levitation Simulation

Medio

Il secondo algoritmo presentato simula un comportamento paranormale. A seguito di un'esplosione, le particelle raggiungono il terreno dal quale comincia la levitazione.L'effetto della forza di levitazione è contrastato dalle rispettive forze peso, la differenza calcolata tra le due forze risulta differente per ogni particella. Tale fenomeno causa un progressivo allontanamento delle particelle tra loro.

Questa simulazione si serve di due tipologie di parametri: **explosion parameters** e **levitation parameters**. La prima tipologia definisce la direzione e l'intensità dell'esplosione iniziale, la seconda è responsabile della forza di levitazione e della sua intensità.

```
// Explosion parameters
vec3f expl_vec  = {1, 1, 1};
float expl_ampl = 2;

// Levitation parameters
vec3f lev_vec   = {0, 1, 0};
float lev_ampl  = 2;

void simulate_levitation(particle_scene& scene, const particle_params&
params) {
  for (auto& shape : scene.shapes) {
    shape.old_positions = shape.positions;
  }
```

```
  rng_state rng = make_rng(params.seed);

  for (auto& shape : scene.shapes) {
    for (int k = 0; k < shape.positions.size(); k++) {
      if (!shape.invmass[k]) continue;
```

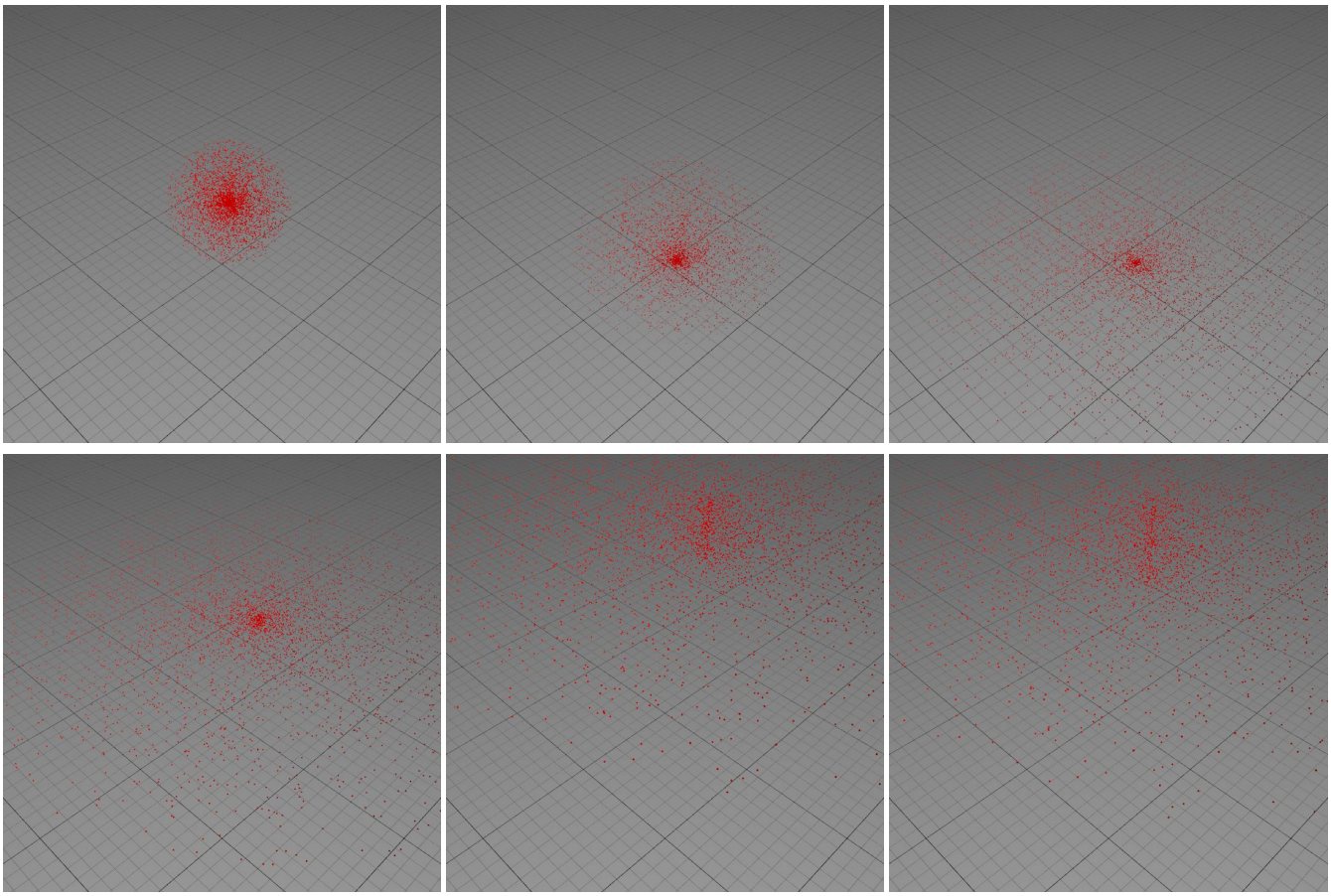
Applicazione delle forze in gioco.

```
      shape.velocities[k] += vec3f{0, -params.gravity * shape.invmass[k],
0} *
                             params.deltat;

      shape.velocities[k] += expl_vec * expl_ampl * params.deltat;
      shape.velocities[k] += lev_vec * lev_ampl * params.deltat;

      shape.positions[k] += shape.velocities[k] * params.deltat;
    }
  }
```

La variazione delle intensità risulta cruciale per ottenere l'effetto desiderato. Nella prima condizione, si ha la decrescita dell'esplosione fino all'intensità nulla. Nella seconda condizione, è possibile osservare la crescita della forza di levitazione che, raggiunto il doppio dell'accelerazione gravitazionale, si imposta nuovamente ad un valore iniziale, generando un'effetto oscillatorio.

```
  if (expl_ampl > 0) {
    expl_ampl -= 0.2;
  }
  if (expl_ampl <= 0) {
    if (lev_ampl >= params.gravity * 1.6) {
      lev_ampl = 4.1;
    } else {
      lev_ampl += 0.1;
    }
  }

  . . .

  // Next istructions are pbd-like
```

Di seguito i risultati ottenuti:



Periodic Functions Application

In questo paragrafo vengono analizzati due algoritmi differenti. Quest'ultimi utilizzano le funzioni periodiche **seno** e **coseno**, per descrivere alcuni effetti oscillatori e rotatori.

Parametric Waves Simulator

Facile

Il simulatore analizzato di seguito applica la funzione coseno con diverse ampiezze, ad un asse fissato in precedenza. Inoltre, le onde possono essere generate con frequenze differenti, variando la velocità angolare.

```
float angle_vib          = 0;
vec3f vibrating_function = {1, 1, 1};
float amplitude          = 500;

void simulate_oscillation(
    particle_scene& scene, const particle_params& params) {
  for (auto& shape : scene.shapes) {
    shape.old_positions = shape.positions;
  }

  rng_state rng = make_rng(params.seed);

  for (auto& shape : scene.shapes) {
    for (int k = 0; k < shape.positions.size(); k++) {
```

Conversione dell'angolo in radianti e applicazione della funzione coseno sull'asse prefissato.

```
      float ang_vib = angle_vib * M_PI / 180;

      if (params.axis == 0) {

        shape.velocities[k].x = 0;
        vibrating_function    = {cos(ang_vib), 0, 0};

      } else if (params.axis == 1) {

        shape.velocities[k].y = 0;
        vibrating_function    = {0, cos(ang_vib), 0};

      } else if (params.axis == 2) {

        shape.velocities[k].z = 0;
        vibrating_function    = {0, 0, cos(ang_vib)};

      }

      if (!shape.invmass[k]) continue;
```

Applicazione della forza oscillatoria.

```
      shape.velocities[k] += vibrating_function * amplitude *
                             normalize(vec3f{1, 1, 1}) * params.deltat;

      shape.positions[k] += shape.velocities[k] * params.deltat;
    }
  }
```

Aggiornamento dell'angolo. Maggiore è il valore di aggiornamento per iterazione, maggiore è la frequenza d'oscillazione.

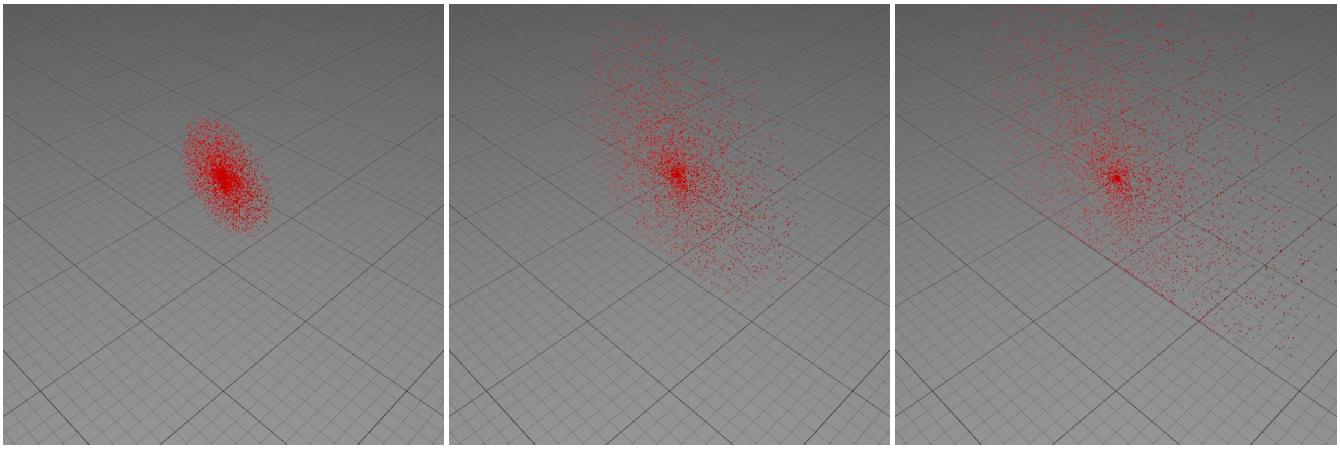
```
  angle_vib += 30;
  if (angle_vib >= 360) {
    angle_vib = 0;
  }
```

Smorzamento dell'oscillazione.

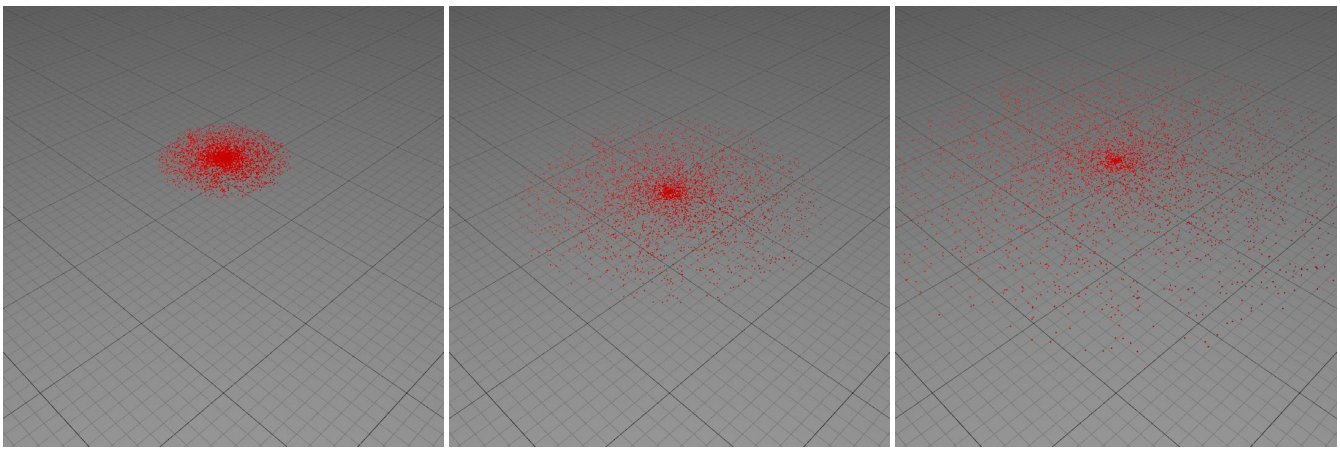
```
    if (amplitude > 0) {
      amplitude -= 1;
    }

    . . .

    // Next istructions are pbd-like
```

Di seguito i risultati ottenuti variando la velocità angolare, l'ampiezza e l'asse d'applicazione.

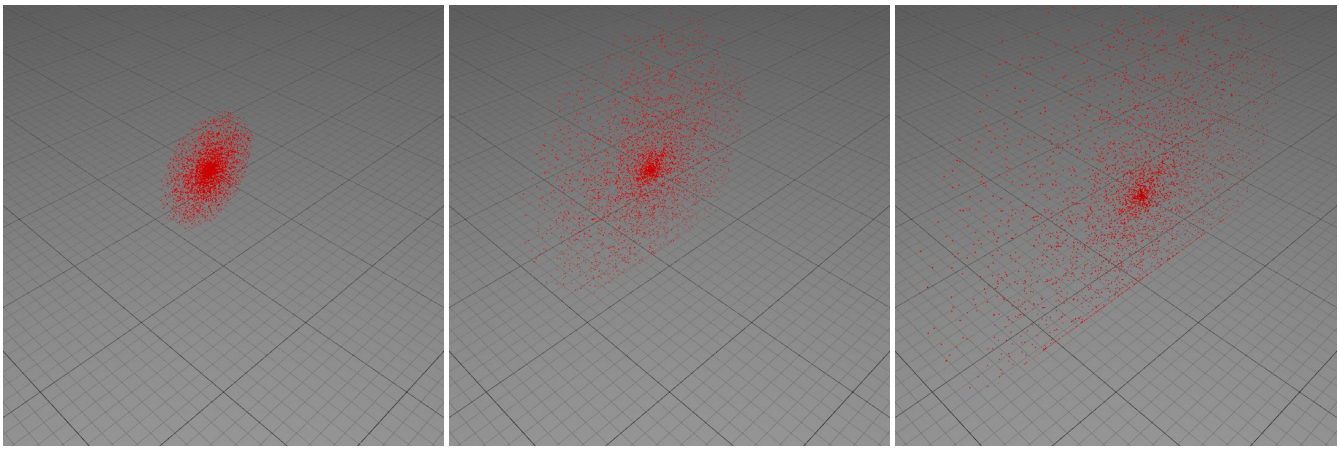
x-axis



y-axis



z-axis



Tornado Simulator

Facile

Nel secondo algoritmo di questo paragrafo viene analizzata l'applicazione delle funzioni periodiche, per la generazione di un moto rotatorio. Quest'ultimo è ottenuto sfruttando tale proprietà: preso un punto in moto circolare, si possono tracciare le sue componenti **x** e **y**, utilizzando rispettivamente **coseno** e **seno**. Nel caso seguento però, la funzione seno viene applicata all'**asse delle quote** per ottenere un moto circolare parallelo al terreno.

La variazione dello sfasamento iniziale del parametro **ang**, ci permette di ottenere inizializzazioni diverse del tornado.

```
float ang     = 220;
vec3f tornado = {0, 0, 0};

void simulate_tornado(particle_scene& scene, const particle_params& params)
{
  for (auto& shape : scene.shapes) {
    shape.old_positions = shape.positions;
  }

  for (auto& shape : scene.shapes) {
    for (int k = 0; k < shape.positions.size(); k++) {
      if (!shape.invmass[k]) continue;
```

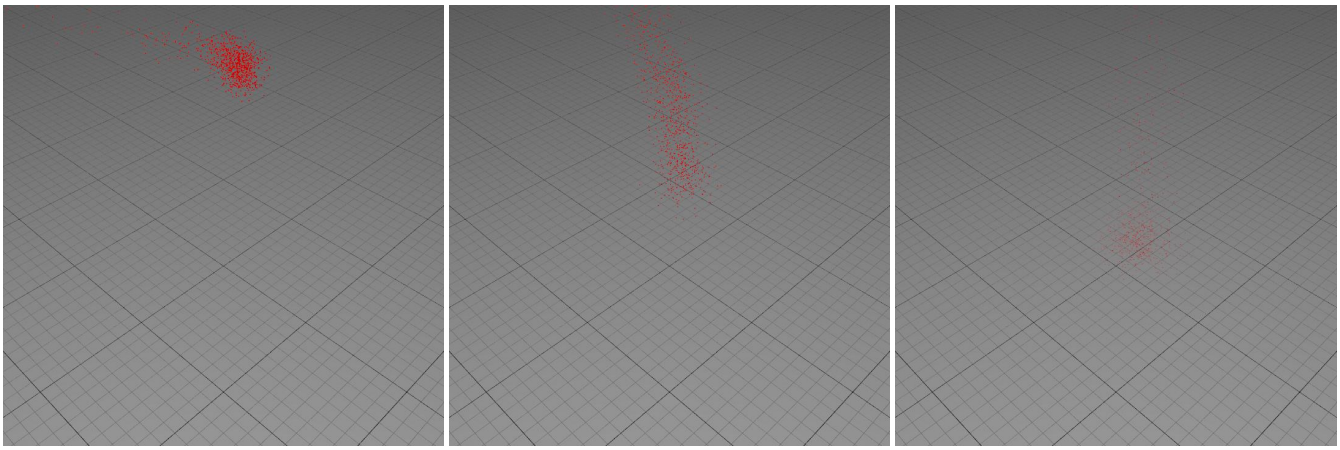
Conversione dell'angolo in radianti e applicazione delle funzioni periodiche agli assi coinvolti.

```
      float radang = ang * M_PI / 180;
      tornado.x    = cos(radang);
      tornado.z    = sin(radang);

      rng_state rng = make_rng(params.seed);
```

Applicazione delle forze in gioco.

```
      shape.velocities[k] +=
          (vec3f{0, -params.gravity * shape.invmass[k] / 2, 0} +
              tornado / shape.invmass[k]) *
          params.deltat;
      shape.positions[k] += shape.velocities[k] * params.deltat;

      if (ang == 360) {
        ang = 0;
      } else {
        ang += 0.006;
      }
    }
  }
```

Di seguito sono riportati i risultato ottenuti:



Bouncing Simulator

Facile

L'algoritmo preso in esame, sfrutta l'applicazione di **Neighbours Set** per generare da una **cloth-like shape** una sfera caratterizzata da **bounciness**.

Il simulatore in questione utilizza una funzione di inizializzazione dedicata: **init_creative_simulation**. Tale funzione genera i **Neighbours Set** per ogni particella, mantenendo l'implementazione delle springs **cloth-like**.

Bounce initializer

```
void init_creative_simulation(
    particle_scene& scene, const particle_params& params) {
  reset_global_variables();
  auto sid = 0;
  for (auto& shape : scene.shapes) {
    shape.emit_rng     = make_rng(params.seed, (sid++) * 2 + 1);
```

```
    shape.invmass      = shape.initial_invmass;
    shape.normals      = shape.initial_normals;
    shape.positions    = shape.initial_positions;
    shape.radius       = shape.initial_radius;
    shape.velocities   = shape.initial_velocities;
    shape.bounce       = params.bouciness;
    shape.has_collided = false;

    shape.forces.clear();
    for (int i = 0; i < shape.positions.size(); i++)
      shape.forces.push_back(zero3f);

    // No pinned particles

    for (auto& velocity : shape.velocities) {
      velocity += sample_sphere(rand2f(shape.emit_rng)) *
shape.emit_rngscale *
                  rand1f(shape.emit_rng);
    }
```

Generazione dei **Neighbours Set**.

```
    for (int i = 0; i < shape.positions.size(); i++) {
      particle_neighbours particle;
      particle.particle = shape.positions[i];
      particle.index    = i;
      scene.groups.push_back(particle);
      for (int j = 0; j < shape.positions.size(); j++) {
        if (i != j && distance(shape.positions[i], shape.positions[j]) <
0.2) {
          scene.groups[i].neighIdxs.push_back(j);
          scene.groups[i].neighbours.push_back(shape.positions[j]);
        }
      }
    }

    shape.springs.clear();

    if (shape.spring_coeff > 0) {
      for (auto& edge : get_edges(shape.quads)) {
        shape.springs.push_back({edge.x, edge.y,
            distance(shape.positions[edge.x], shape.positions[edge.y]),
            shape.spring_coeff});
      }

      for (auto& quad : shape.quads) {
        shape.springs.push_back({quad.x, quad.z,
            distance(shape.positions[quad.x], shape.positions[quad.z]),
            shape.spring_coeff});
        shape.springs.push_back({quad.y, quad.w,
            distance(shape.positions[quad.y], shape.positions[quad.w]),
```

```
            shape.spring_coeff});
      }
    }
  }

  for (auto& collider : scene.colliders) {
    collider.bvh = {};

    if (collider.quads.size() > 0)
      collider.bvh = make_quads_bvh(
          collider.quads, collider.positions, collider.radius);
    else
      collider.bvh = make_triangles_bvh(
          collider.triangles, collider.positions, collider.radius);
  }
}
```

Bounce Simulator

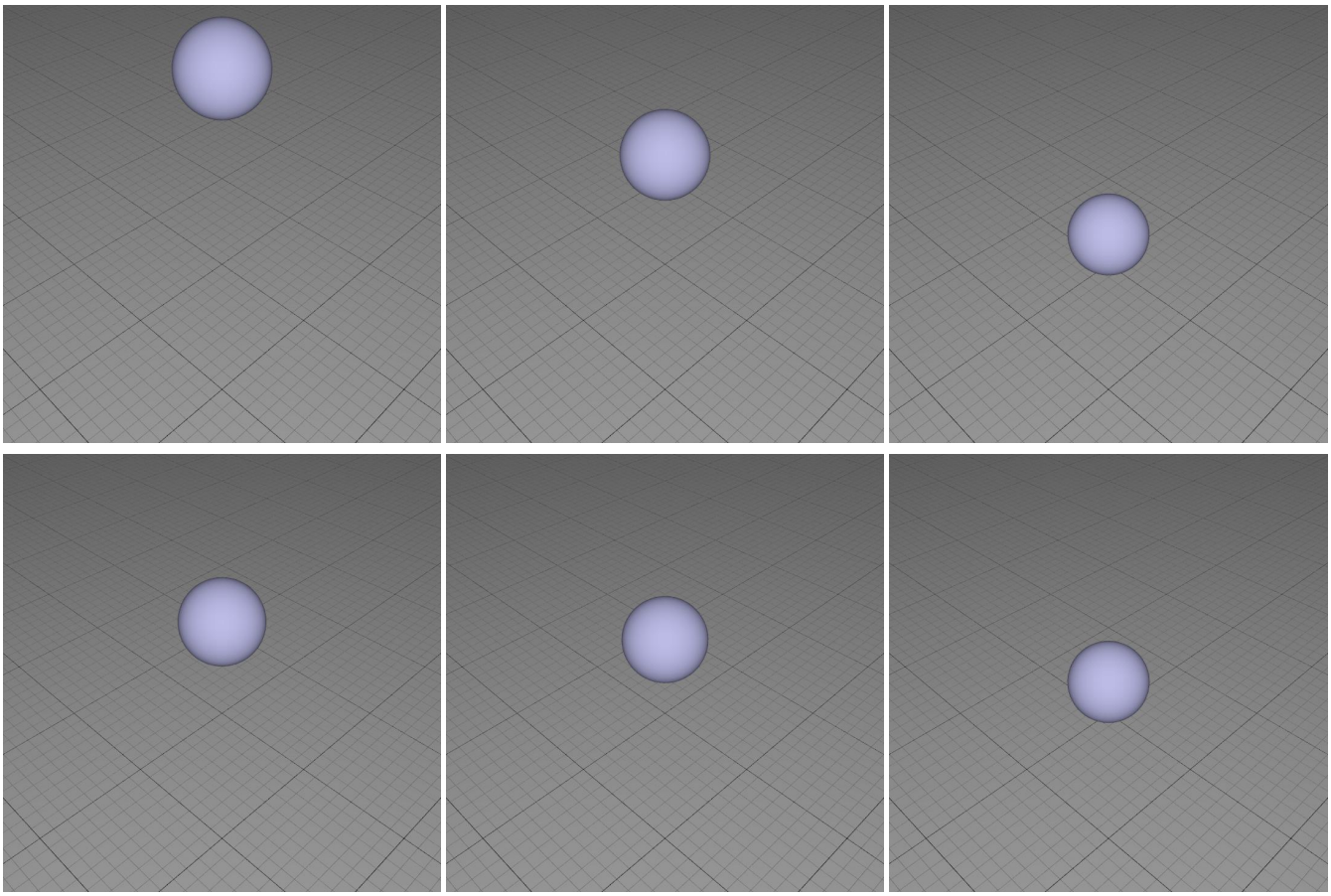
Grazie alla generazione dei **Neighbours Set**, il simulatore potrà applicare le forze di collisione non solo alla particelle in questione ma anche alle particelle limitrofe. Di seguito è riportato il segmento del simulatore responsabile della risposta alle collisioni.

```
void simulate_bounce(particle_scene& scene, const particle_params&  params)
{

      // Previous istructions are pbd-like

      . . .

      for (auto& collision : shape.collisions) {
        auto& particle = shape.positions[collision.vert];
        if (!shape.invmass[collision.vert]) continue;
        auto projection = dot(particle - collision.position,
collision.normal);
        if (projection >= 0) continue;
```

La condizione espressa di seguito permette di ricreare la **bounciness** della sfera. Nel primo caso la velocità delle particelle in collisione è ancora maggiore di 0.1, di conseguenza la sfera continua a collidere. Nel secondo caso la sfera è stata quasi fermata dalla collisione, quindi risponde con una forza repulsiva uguale e contraria.

```
        if (length(shape.velocities[collision.vert]) > 0.1) {
          float density = 0.01f;

          particle += -projection * collision.normal * density;

          for (int index : scene.groups[collision.vert].neighIdxs) {
```

```
            shape.positions[index] += -projection * collision.normal *
density;
          }
        } else {
          shape.velocities[collision.vert] = vec3f{0, shape.bounce, 0} *
                                             params.deltat;
          particle += shape.velocities[collision.vert] * params.deltat;

          for (int index : scene.groups[collision.vert].neighIdxs) {
            shape.velocities[index] += vec3f{0, shape.bounce, 0} *
                                       collision.normal * params.deltat;
            shape.positions[index] += shape.velocities[index] *
params.deltat;
          }
          shape.bounce -= 50;
        }
      }
    }
  }

  . . .

  // Next istructions are pbd-like
```



Parametric Wind

Molto facile

L'ultimo simulatore presentato in questo paper applica ad una **cloth-like shape** un forza configurabile. Quest'ultima, in aggiunta all'accelerazione gravitazionale, simula l'applicazione di un vento con diverse intensità e direzioni.

```
vec3f wind = {0, 0, 0};

void simulate_wind(particle_scene& scene, const particle_params& params) {

  for (auto& shape : scene.shapes) {
    shape.old_positions = shape.positions;
  }
```

Al momento dell'esecuzione vengono aggiornate le coordinate del vento, impostate dall'utente. Quest'ultime generano il vettore **wind** che, durante l'esecuzione, applica una forza all'oggetto.

```
  for (auto& shape : scene.shapes) {
    for (int k = 0; k < shape.positions.size(); k++) {
      if (!shape.invmass[k]) continue;

      wind.x = params.xwind;
      wind.y = params.ywind;
      wind.z = params.zwind;

      shape.velocities[k] += (vec3f{0, -params.gravity, 0} + wind) *
                             params.deltat;
      shape.positions[k] += shape.velocities[k] * params.deltat;
    }
  }
  . . .

  // Next istructions are pbd-like
```