Particle Systems

Particle Systems

Why we need them

What they can do

How to do it

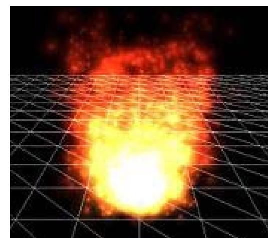
Particle Systems

A very large number of very tiny particles

- May or may not have any geometry

Useful for:

- Fire, explosions, gas, clouds, fireworks
- Anything where something breaks into a zillion pieces



A Particle

```
struct Particle
{
    CGrPoint    p;          // Where is it?
};

std::list<Particle> m_particles;       // All of our particles
```

How to render

We have two choices:

Particles with geometry

- Means larger than a pixel
- Not very common, mostly early work

Particles with no geometry

- Just a tiny point in space
- We'll concentrate on these

Rendering a particle in OpenGL

The gist of it:

- Don't
- Yes, OpenGL can render a "point", but it's an entire pixel. It's in general a bad idea

How to render

- Create a image
- Render into the image

Creating an Image

We need an RGB image

- Just like in the ray tracer
- But, we're going to make it float value instead of byte
  - We're going to do math on the pixel multiple times

Creating the Image

```
m_partimage = new float *[m_partimageheight];
m_partimage[0] = new float[m_partimageheight * m_partimagewidth * 3];

for(int i=1; i<m_partimageheight; i++)
{
    m_partimage[i] = m_partimage[0] + i * m_partimagewidth * 3;
}


for(int i=0; i<m_partimageheight; i++)
{
   // Fill the image with black
   for(int j=0; j<m_partimagewidth; j++)
   {
      m_partimage[i][j * 3] = 0;          // red
      m_partimage[i][j * 3 + 1] = 0;      // green
      m_partimage[i][j * 3 + 2] = 0;      // blue
   }
}
```

Rendering a Particle

We need to convert the X,Y,Z coordinate to a screen location

- This is projection, just as we did before
  - Remember how to do this?
- We want to mix this with OpenGL, so we'll use their matrices and values

Projection

Multiply point by model view matrix

Then by projection matrix

Homogenize

Convert to viewport

Stealing from OpenGL

```
GLdouble m[16];
GLdouble *pm;

// Obtain the ModelView matrix and convert it to CGrTransform
glGetDoublev(GL_MODELVIEW_MATRIX, m);
CGrTransform modelViewMatrix;
pm = m;
for(int c=0;  c<4;  c++)
    for(int r=0;  r<4;  r++)
        modelViewMatrix[r][c] = *pm++;

// Obtain the projection matrix and convert it to CGrTransform
glGetDoublev(GL_PROJECTION_MATRIX, m);
CGrTransform projectionMatrix;
pm = m;
for(int c=0;  c<4;  c++)
    for(int r=0;  r<4;  r++)
        projectionMatrix[r][c] = *pm++;

// We can go ahead and multiply these together
m_partxform = projectionMatrix * modelViewMatrix;

// Obtain the glViewport parameters
glGetIntegerv(GL_VIEWPORT, m_vp);
```


Particle Projection

```
// Multiply by the matrices
CGrPoint p1 = m_partxform * p.p;

// Test for content behind us
if(p1[2] <= 0)
    return;        // Don't draw

// Homogenize
p1[0] /= p1[3];
p1[1] /= p1[3];

// Viewport conversion
int sx = int((p1[0] + 1) * m_vp[2] / 2 + m_vp[0]);
int sy = int((p1[1] + 1) * m_vp[3] / 2 + m_vp[1]);

// Clip anything that is off the screen
if(sx < 0 || sx >= m_partimagewidth || sy < 0 || sy >= m_partimageheight)
    return;

// And add it to the pixel
m_partimage[sy][sx * 3] += p.color[0];
m_partimage[sy][sx * 3 + 1] += p.color[1];
m_partimage[sy][sx * 3 + 2] += p.color[2];
```

Notice: This version does not care about occlusion

What do to with the pixel

Add to pixel (simplest)

```
m_partimage[sy][sx * 3] += p.color[0];
m_partimage[sy][sx * 3 + 1] += p.color[1];
m_partimage[sy][sx * 3 + 2] += p.color[2];
```

Replace pixel

```
m_partimage[sy][sx * 3] = p.color[0];
m_partimage[sy][sx * 3 + 1] = p.color[1];
m_partimage[sy][sx * 3 + 2] = p.color[2];
```

Translucent

```
double translucency = 0.5;
m_partimage[sy][sx * 3] = translucency * m_partimage[sy][sx * 3] + p.color[0];
m_partimage[sy][sx * 3 + 1] = translucency * m_partimage[sy][sx * 3 + 1] + p.color[0];
m_partimage[sy][sx * 3 + 2] = translucency * m_partimage[sy][sx * 3 + 2] + p.color[0];
```


What about occlusion

What do I mean by occlusion?

Is it an issue?

What could we do about it?

Handling Occlusion

```
struct Particle
  {
     Particle() : p(0, 0, 0) {}

     CGrPoint    p;          // Where is it?
     float       color[3];   // The particle color

     CGrPoint    projected;  // Particle after projection matrix

     bool operator<(const Particle &p) {return projected[2] > p.projected[2];}
  };
```

We'll project everything, sort, then render the particles in back to front order

z value after projection

Projection and Sorting

```
// Project all of my particles
for(std::list<Particle>::iterator p=m_particles.begin();  p!=m_particles.end();  p++)
{
   p->projected = m_partxform * p->p;
}

m_particles.sort();

// Render all of the particles
for(std::list<Particle>::iterator p=m_particles.begin();  p!=m_particles.end();  p++)
{
   ParticleRender(*p);
}
```

Rendering

```
void CParticles::ParticleRender(const Particle &p)
{
    // Test for content behind us
    if(p.projected[2] <= 0)
        return;          // Don't draw

    // Homogenize
    double x = p.projected[0] / p.projected[3];
    double y = p.projected[1] / p.projected[3];

    // Viewport conversion
    int sx = int((x + 1) * m_vp[2] / 2 + m_vp[0]);
    int sy = int((y + 1) * m_vp[3] / 2 + m_vp[1]);

    // Clip anything that is off the screen
    if(sx < 0 || sx >= m_partimagewidth || sy < 0 || sy >= m_partimageheight)
        return;

    // And add it to the pixel
    float translucency = 0; // 0.5f;
    m_partimage[sy][sx * 3] = translucency * m_partimage[sy][sx * 3] + p.color[0];
    m_partimage[sy][sx * 3 + 1] = translucency * m_partimage[sy][sx * 3 + 1] + p.color[1];
    m_partimage[sy][sx * 3 + 2] = translucency * m_partimage[sy][sx * 3 + 2] + p.color[2];
}
```


What we have

Particles have a location in space

Particles have a color

- (Just assigned for now)

We project, sort, and render

What next?

Creating Particles

Simple demo just creates like this:

```
void CChildView::OnLoadSquare()
{
   m_particles.Clear();

   for(int i=0;  i<20000;  i++)
   {
      CGrPoint p(random(-5, 5),       // x
               random(25, 35),      // y
               random(-5, 5));      // z

      float color[] = {(float)random(0, 0.5), // r
                  (float)random(0, 0.5),   // g
                  (float)random(0, 0.5) }; // b

       m_particles.LoadParticle(p, color);
   }
}
```


Options for creating particles

Up front

- Random locations inside a volume
  - Might be a mesh
- Emitted at certain locations
- Vertices of object

Continuous

- Emitted from certain location
- Emitted from a geometry

What to do with them?

Static particles are boring
- Why bother

What makes particles cool is motion
- Usually simple physics

Particles

Particles are objects modeled as point masses

Particle properties:
- mass
- position
- velocity
- force accumulator
- age, lifespan
- rendering properties
- etc.

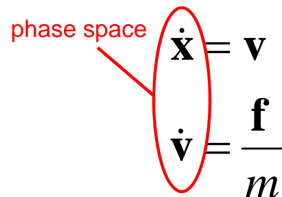
Particles

Particles respond to forces

We represent this using differential equations

$$\ddot{\mathbf{x}} = \frac{\mathbf{f}}{m}$$

2nd order ODE



$$\dot{\mathbf{x}} = \mathbf{v}$$

$$\dot{\mathbf{v}} = \frac{\mathbf{f}}{m}$$

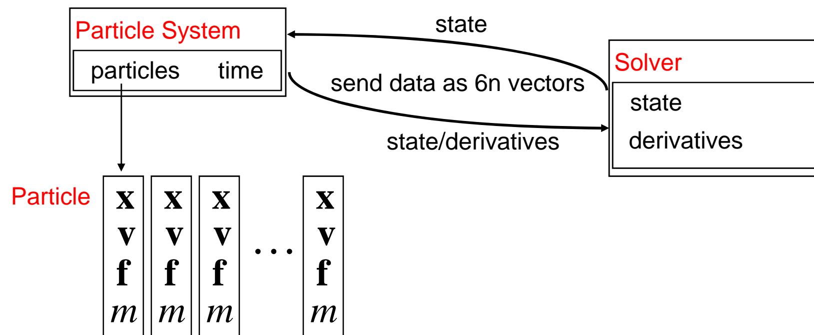
1st order ODEs

Particle with simple physics

```
struct Particle
  {
    Particle() : p(0, 0, 0), v(0, 0, 0, 0) {}

    CGrPoint    p;         // Where is it?
    CGrPoint    v;         // Velocity

    float      color[3];   // The particle color

    CGrPoint    projected;  // Particle after projection matrix

    bool operator<(const Particle &p) {return projected[2] > p.projected[2];}
  };
```

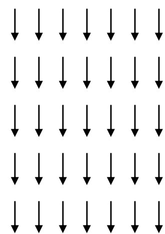
Particle Systems

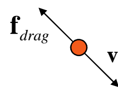
Separate the data structures and integration

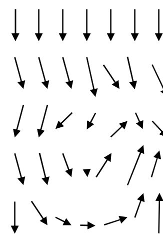


Forces

Unary forces -
forces that only depend on 1 particle







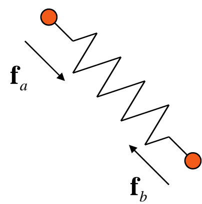
Gravity $\mathbf{f} = m\mathbf{g}$

Dampening $\mathbf{f}_{drag} = -k_d \mathbf{v}$

Wind Fields $\mathbf{f} = k\,\mathbf{v}_{wind}$

Forces

Binary forces -
forces that only depend on 2 particles



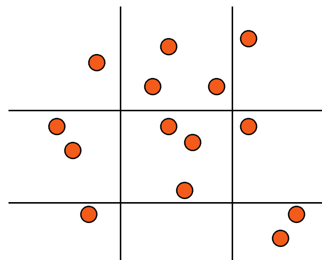
Springs

$$\mathbf{f}_a = -k_s(|\mathbf{x}_a - \mathbf{x}_b| - l_0)\frac{\mathbf{x}_a - \mathbf{x}_b}{|\mathbf{x}_a - \mathbf{x}_b|} - k_d\left(\frac{(\mathbf{v}_a - \mathbf{v}_b)\cdot(\mathbf{x}_a - \mathbf{x}_b)}{|\mathbf{x}_a - \mathbf{x}_b|}\right)\frac{\mathbf{x}_a - \mathbf{x}_b}{|\mathbf{x}_a - \mathbf{x}_b|}$$

Generalizes to n-ary forces

Forces

Spatial forces -
forces that depend on *local* particles

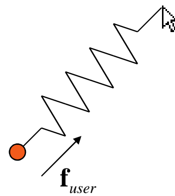


(Gravity), Lennard-Jones and electric potentials

Spatial data structures can optimize computations

Forces

User Interaction forces -
forces applied to particles by the user



Usually just use springs

Soft Constraints

It is often easier to specify a condition that we want to be true rather than a force to make it true

Constraint (or behavior) functions
specify a condition that we wish to have fulfilled

$$\mathbf{C}(\mathbf{x}_a, \mathbf{x}_b) = \mathbf{x}_a - \mathbf{x}_b = \mathbf{0}$$

or

$$C(\mathbf{x}_a, \mathbf{x}_b) = |\mathbf{x}_a - \mathbf{x}_b| - r = 0$$

Constraint functions

But how do we make it true?

Energy Functions

Minimize the energy of the constraint

$$E = \frac{k_s}{2} \mathbf{C} \cdot \mathbf{C}$$

Force the particle towards the constraint

$$\mathbf{f}_i = -\frac{\partial E}{\partial \mathbf{x}_i} = -k_s \frac{\partial \mathbf{C}}{\partial \mathbf{x}_i} \mathbf{C}$$

$$\mathbf{f}_i^{dampening} = -k_d \frac{\partial \mathbf{C}}{\partial \mathbf{x}_i} \dot{\mathbf{C}}$$

Energy Functions

Example $\mathbf{C} = \mathbf{x}_a - \mathbf{x}_b$

$\frac{\partial \mathbf{C}}{\partial \mathbf{x}_a} = \mathbf{I}$	$\frac{\partial \mathbf{C}}{\partial \mathbf{x}_b} = -\mathbf{I}$	$\dot{\mathbf{C}} = \mathbf{v}_a - \mathbf{v}_b$
x_a derivative	x_b derivative	time derivative

Plug into the force equation

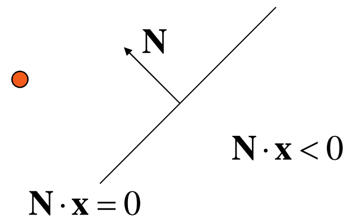
$$\mathbf{f}_i = -k_s \frac{\partial \mathbf{C}}{\partial \mathbf{x}_i} \mathbf{C} - k_d \frac{\partial \mathbf{C}}{\partial \mathbf{x}_i} \dot{\mathbf{C}}$$

$$\mathbf{f}_a = -k_s (\mathbf{x}_a - \mathbf{x}_b) - k_d (\mathbf{v}_a - \mathbf{v}_b)$$

Just a damped spring

Collision Detection

Determine when a particle has collided



Particle has collided iff $\mathbf{N}\cdot\mathbf{x}<0$

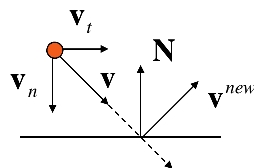
Use raytracing for more complex shapes

Collision Response

What should we do when a particle has collided?

The correct thing to do is rollback the simulation to the exact point of contact

Easier to just modify positions and velocities



After the collision:

$$\mathbf{v}^{new} = -\varepsilon\mathbf{v}_n + \mathbf{v}_t$$

coefficient of restitution

Contact Forces

When the particle is on the collision surface a contact force resists penetration

$$\mathbf{f}^c = -(\mathbf{N}\cdot\mathbf{f})\,\mathbf{f} \qquad (\mathbf{N}\cdot\mathbf{f}) < 0$$

Contact forces do not resist leaving the surface

$$\mathbf{f}^c = 0 \qquad (\mathbf{N}\cdot\mathbf{f}) > 0$$

Simple friction can be modeled

$$\mathbf{f}^f = -k_f(-\mathbf{N}\cdot\mathbf{f})\,v_t \qquad (\mathbf{N}\cdot\mathbf{f}) < 0$$

Integration

Given the differential equations and a starting point (known as an initial value problem)

$$\dot{\mathbf{x}} = \mathbf{v} \qquad \mathbf{x}(t_0) = \mathbf{x}_0$$

$$\dot{\mathbf{v}} = \frac{\mathbf{f}}{m} \qquad \mathbf{v}(t_0) = \mathbf{v}_0$$

We need to integrate to find the new position

$$\mathbf{x}(t_1) = \int_{t_0}^{t_1} \dot{\mathbf{x}}\,dt + \mathbf{x}(t_0)$$

Euler Integration

First order approximation

given $\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}, t)$ and $\mathbf{x}(t_i)$

Taylor expansion gives

$$\mathbf{x}(t_i + dt) = \mathbf{x}(t_i) + \dot{\mathbf{x}}(t_i)\, dt + O(dt^2)$$

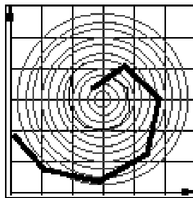
$$\mathbf{x}(t_i + dt) \approx \mathbf{x}(t_i) + \dot{\mathbf{x}}(t_i)\, dt$$

$$\mathbf{x}(t_i + dt) \approx \mathbf{x}(t_i) + \mathbf{f}(\mathbf{x}(t_i), t_i)\, dt$$

Assumes that the force is constant over the timestep

Euler Integration

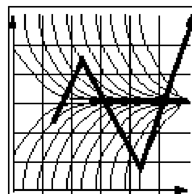
Inaccurate



smaller timesteps only slow the problems down

Unstable

$$\mathbf{f}(x, t) = -k\,\mathbf{x}$$



reasonable if $dt < 1/k$

small timesteps yield bad performance

Midpoint Method

Second order approximation

$$\mathbf{x}(t_i + dt) = \mathbf{x}(t_i) + \dot{\mathbf{x}}(t_i)\, dt + \ddot{\mathbf{x}}(t_i)\, \tfrac{dt^2}{2} + O(dt^3)$$

with $\ddot{\mathbf{x}}(t_i) = \dfrac{\partial \mathbf{f}}{\partial \mathbf{x}} \dfrac{d\mathbf{x}}{dt} = \mathbf{f}'\,\mathbf{f}$

$$\mathbf{f}(\mathbf{x}_i + \Delta\mathbf{x}) = \mathbf{f}(\mathbf{x}_i) + \Delta\mathbf{x}\, \mathbf{f}'(\mathbf{x}_i) + O(\Delta\mathbf{x}^2)$$

$$\mathbf{f}(\mathbf{x}_i + \tfrac{dt}{2}\mathbf{f}(\mathbf{x}_i)) = \mathbf{f}(\mathbf{x}_i) + \tfrac{dt}{2}\mathbf{f}(\mathbf{x}_i)\,\mathbf{f}'(\mathbf{x}_i) + O(dt^2)$$

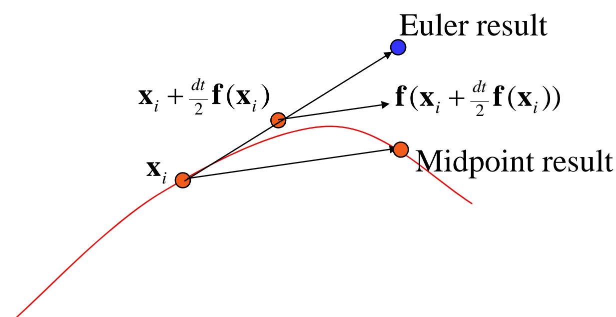
$$\tfrac{dt^2}{2}\ddot{\mathbf{x}} + O(dt^3) = dt\,(\mathbf{f}(\mathbf{x}_i + \tfrac{dt}{2}\mathbf{f}(\mathbf{x}_i)) - \mathbf{f}(\mathbf{x}_i))$$

$$\boxed{\mathbf{x}(t_i + dt) = \mathbf{x}(t_i) + dt\,(\mathbf{f}(\mathbf{x}_i + \tfrac{dt}{2}\mathbf{f}(\mathbf{x}_i)))}$$

Midpoint Method

Geometric Interpretation

$$\mathbf{x}(t_i + dt) = \mathbf{x}(t_i) + dt\,(\mathbf{f}(\mathbf{x}_i + \tfrac{dt}{2}\mathbf{f}(\mathbf{x}_i)))$$



we can use higher orders (ex: runge-kutta 4)

Adaptive Timestepping

Allow the timestep to change over the course of the solution

$$e = |\mathbf{x}_a - \mathbf{x}_b|$$

1 timestep

2 timesteps

Scale the timestep based on the error

$$s = \left(\frac{e^{desired}}{e^{current}} \right)^{-\text{integrationOrder}}$$

Euler Step for Position

Very simple:

- p = p + Δv

```
void CParticles::Update(double delta)
{
   for(std::list<Particle>::iterator p=m_particles.begin();  p!=m_particles.end();  p++)
   {
      // Euler step
      p->p += p->v * delta;

      // Ensure W value remains 1
      p->p.W(1);
   }

}
```

Loading A Cube with Velocity

```
void CChildView::OnLoadSquarewithrandomvelocities()
{
   m_particles.Clear();

   float red[] = {1, 0, 0};
   float grn[] = {0, 1, 0};

   for(int i=0;  i<20000;  i++)
   {
      CGrPoint p(random(-5, 5),       // x
               random(25, 35),      // y
               random(-5, 5));      // z

      float color[] = {(float)random(0, 0.5), // r
                  (float)random(0, 0.5),   // g
                  (float)random(0, 0.5) }; // b

      CGrPoint v(random(10, 15),      // x
               random(-5, 5),       // y
               random(-5, 5));      // z

      m_particles.LoadParticle(p, v, color);
   }
}
```


Acceleration

To complete our simply physics

- Acceleration is the derivative of velocity

Euler step for velocity: $v = \Delta a$

```
for(std::list<Particle>::iterator p=m_particles.begin();  p!=m_particles.end();  p++)
  {
     CGrPoint a(0, 0, 0, 0);     // Acceleration

     if(m_gravity)
     {
        a.Y(-9.80665);        // m/sec
     }

     // Euler steps
     p->v += a * delta;
     p->p += p->v * delta;

     p->p.W(1);
  }
```

Bounce

What happens when we hit the ground?

We bound in the bounce direction

- Any ideas on that equation?

Just change the velocity to the bounce direction

But, we need to know when we hit!

Bounce (reflection) direction

$$V' = -2(N \cdot V)N + V$$

Same equation as reflection direction, except V is negated

First approximation

```
if(p->p.Y() < 0)
{
   CGrPoint N(0, 1, 0, 0);     // Straight up
   p->v = N * -2 * Dot3(N, p->v) + p->v;
}
```


Problems with this version

We may not have bounced back above zero

- We only consider us under the floor if we are still going down

Better Approximation

```
if(p->p.Y() < 0 && p->v.Y() < 0)
{
   CGrPoint N(0, 1, 0, 0);     // Straight up
   p->v = N * -2 * Dot3(N, p->v) + p->v;
}
```


Another Problem

We're not really bouncing on the floor

- What are we doing?

We need to know the time to hit the floor!

Determining a Collision Time

```
// This is how much we need to accomplish
double deltaToDo = delta;

// Keep trying until we get it done
while(deltaToDo > 0)
{
    double deltaTry = deltaToDo;

    while(true)
    {
        if(TryUpdate(*p, deltaTry))
        {
            deltaToDo -= deltaTry;
            break;
        }

        deltaTry /= 2;      // Try half as much
    }

}
```

```
bool CParticles::TryUpdate(Particle &particle, double delta)
{
    CGrPoint a(0, 0, 0, 0);     // Acceleration

    if(m_gravity)
    {
        a.Y(-9.80665);        // m/sec
    }

    // Euler steps
    CGrPoint v = particle.v + a * delta;
    CGrPoint p = particle.p + v * delta;

    // Are we too far below?
    if(p.Y() < -0.000001)
        return false;

    if(p.Y() < 0 && v.Y() < 0)
    {
        CGrPoint N(0, 1, 0, 0);     // Straight up
        v = N * -2 * Dot3(N, v) + v;
    }

    particle.v = v;
    p.W(1);
    particle.p = p;

    return true;
}
```

What is wrong with this, now?

Loss

Really, we expect to lose energy when we bounce.

- Easy enough:

```
v = N * -2 * Dot3(N, v) + v;
v = v * 0.75;
```

Complications

This works fine until the particles stop

- Add gravity only if particle above zero or moving up
- Only count as bad a drop if drops from “above” to “below”
- Force to intersection point at bounce
  - (Y = 0 in our example)


```
bool CParticles::TryUpdate(Particle &particle, double delta)
{
   CGrPoint a(0, 0, 0, 0);     // Acceleration

   if(m_gravity && (particle.p.Y() > 0 || particle.v.Y() > 0))
   {
      a.Y(-9.80665);       // m/sec
   }

   // Euler steps
   CGrPoint v = particle.v + a * delta;
   CGrPoint p = particle.p + v * delta;

   // Are we too far below?
   if(particle.p.Y() > 0 && p.Y() < -0.00000001)
         return false;

   if(p.Y() < 0 && v.Y() < 0)
   {
      CGrPoint N(0, 1, 0, 0);     // Straight up
      v = N * -2 * Dot3(N, v) + v;
      v = v * 0.75;
      p.Y(0);
   }

   particle.v = v;
   p.W(1);
   particle.p = p;

   return true;
}
```

Now for the real fun

Randomizing

- Randomize the bounce
  - How much loss
  - Varying direction
- Randomize start velocity
- Randomized emission

Energy

It's common to add an energy variable to a particle

- Energy decays with time
- When below a certain point, particle disappears
- Use energy to determine color (hue)

Other tricks

Treat particle path as a line

- Called strands or hairs

Particles can spawn new particles