# Quiz 2

**Take-home**

**5pm Wednesday, Nov. 30 – 2am Thursday, Dec. 1**

# Procedural methods

# Procedural Methods

**How can we model**

- Natural phenomena
  - Clouds
  - Terrain
  - Plants
- Crowd Scenes
- Real physical processes

**Procedural methods:**

Describe objects in an algorithmic way and generate polygons when needed during rendering

# Procedural Approaches

- **Physically-based models and particle system**
  - Describing dynamic behaviors
    - Fireworks
    - Flocking behavior of birds
    - Wave action
- **Language-based models**
  - Describing trees or terrain
  - Representing relationships
- **Fractal geometry**

# Newtonian Particle

**Particle system is a set of particles**

**Each particle is an ideal point mass**

- Gives the positions of particles
- At each location, we can show an object

**Six degrees of freedom**

- Position
- Velocity

**Each particle obeys Newtons' law**

$$\mathbf{f} = m\mathbf{a}$$

Vectors in 3D

# Particle Equations

The state of the i^th particle is defined by its position $\mathbf{p}_i = \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix}$

Then, we have 6 ordinary differential equations:

Velocity $\mathbf{v}_i = \frac{d\mathbf{p}_i}{dt} = \begin{bmatrix} \frac{dx_i}{dt} \\ \frac{dy_i}{dt} \\ \frac{dz_i}{dt} \end{bmatrix}$

Acceleration $\mathbf{a}_i = \frac{d\mathbf{v}_i}{dt} = \frac{1}{m_i}\mathbf{f}_i(t)$

The question is how we get the force vector

# Solution of Particle Systems

//For a system with *n* particles

```
float time, delta, state[6n], force[3n];
state = initial_state();
for(time = t0; time<final_time, time+=delta) {
        //compute forces
        force =  force_function(state, time);
        // solve the differential equation
        state = ode(force, state, time, delta);
        render(state, time)
}
```

# Force Vector

**Depending on how particles interact with each other**

- Independent Particles O(n)
  - Gravity
  - Drag
- Coupled Particles O(n)
  - Spring-Mass Systems
  - Meshes
- Coupled Particles O($n^2$)
  - Attractive and repulsive forces

# Simple Forces

**Consider force on a particle i**

$$\mathbf{f}_i = \mathbf{f}_i(\mathbf{p}_i, \mathbf{v}_i)$$

**Gravity** $\mathbf{f}_{gi} = m_i\,\mathbf{g}$

$$\mathbf{g}_i = (0, -g, 0)$$

**Drag** $\mathbf{f}_{di} = \mu_i\,\mathbf{f}_{normi}$

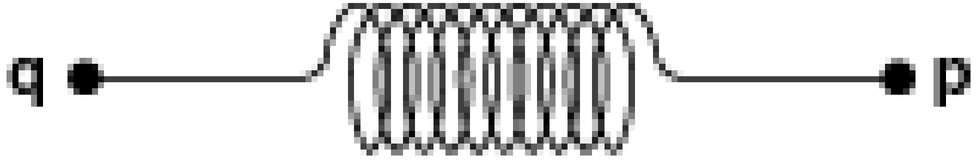
$\mathbf{p}_i(t_0)$, $\mathbf{v}_i(t_0)$

# Spring Forces

Assume each particle has unit mass and is connected to its neighbor(s) by a spring

Keep particles together

Hooke's law: force proportional to distance ($d = \|p - q\|$) between the points



# Hooke's Law

Let $s$ be the distance when there is no force (resting length)

The force is acted on **p** from **q**

$$\mathbf{f} = -k_s(|\mathbf{d}| - s)\,\frac{\mathbf{d}}{|\mathbf{d}|}$$

$k_s$ is the spring constant

$\frac{\mathbf{d}}{|\mathbf{d}|}$ is a unit vector pointed from $p$ to $q$

# Spring Damping

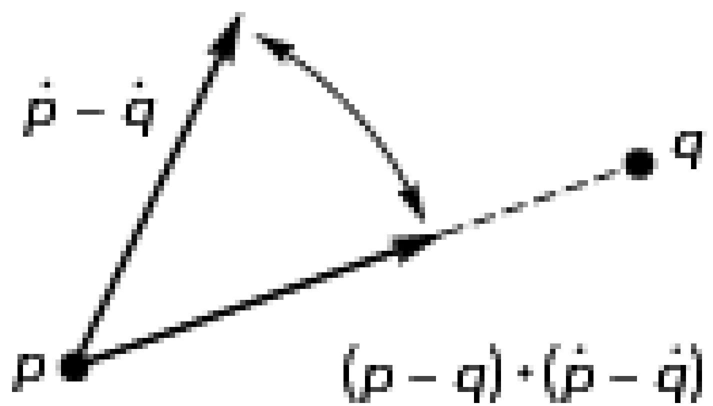
A pure spring-mass will oscillate forever

Must add a damping term

$$\mathbf{f} = -\left(k_s(|\mathbf{d}| - s) + k_d \frac{\dot{\mathbf{d}} \cdot \mathbf{d}}{|\mathbf{d}|}\right)\frac{\mathbf{d}}{|\mathbf{d}|}$$

Damping constant

$$\dot{\mathbf{d}} = \dot{\mathbf{p}} - \dot{\mathbf{q}}$$
$$\dot{\mathbf{d}} \cdot \mathbf{d} = (\dot{\mathbf{p}} - \dot{\mathbf{q}}) \cdot (\mathbf{p} - \mathbf{q})$$



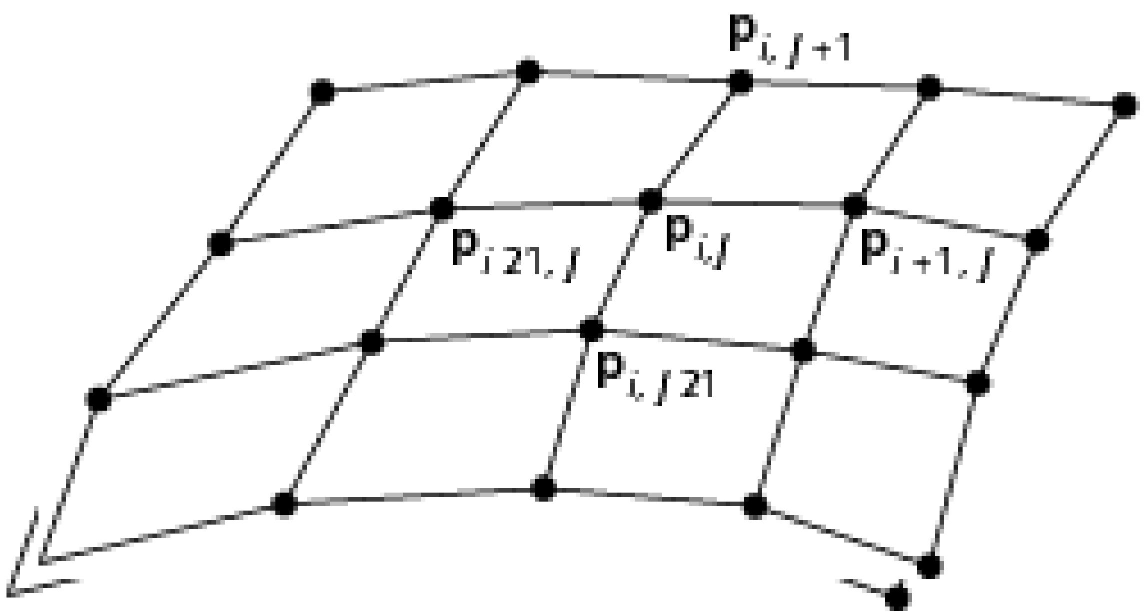
# Meshes

**Connect each particle to its closest neighbors**

- O(n) force calculation

**Use spring-mass system**

Each interior point in mesh has four forces applied to it



# Attraction and Repulsion

***Attraction forces*** pull particles toward each other

***Repulsion forces*** push particles away from each other

- Distribute objects
- Keep objects from hitting each other

These two types of forces are the same except for a sign

# Attraction and Repulsion

For a pair of particles at **p** and **q**

Inverse square law

$$\mathbf{f} = -k_r \frac{\mathbf{d}}{|\mathbf{d}|^3}$$

General case requires O($n^2$) calculation

In most problems, the drop off is such that not many particles contribute to the forces on any given particle

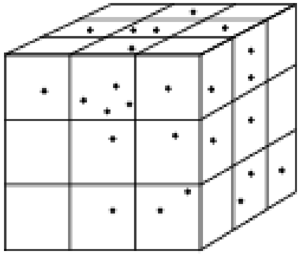
# Boxes

Spatial subdivision technique

Divide space into boxes

Particle can only interact with particles in its box or the neighboring boxes

Must update which box a particle belongs to after each time step



# Linked Lists

Each particle maintains a linked list of its neighbors

Update data structure at each time step

# Angel's Example of Particles in a Box

```
float forces( int i, int j )
{
    int k;
    float force = 0.0;
    /* simple gravity */
    if ( gravity && j == 1 )
        force = -1.0;
    /* repulsive force */
    if ( repulsion )
        for ( k = 0; k < num_particles; k++ ) {
            if ( k != i )
                force +=  0.001 * ( particles[i].position[j] - particles[k].position[j] ) / ( 0.001 +
d2[i][k] );
        }
    return ( force );
}
```

# Example (Cont'd)

```
void idle( void )
{
    int i, j, k;
    float  dt;
    present_time = glutGet( GLUT_ELAPSED_TIME );
    dt = 0.001 * ( present_time - last_time );
    for ( i = 0; i < num_particles; i++ ) {
        for ( j = 0; j < 3; j++ ) {
            particles[i].position[j] += dt * particles[i].velocity[j];
            particles[i].velocity[j] += dt * forces( i, j ) / particles[i].mass;
        }
        collision( i );
    }
…
```

# Example (Cont'd)

```
…
if ( repulsion )
        for ( i = 0; i < num_particles; i++ )
            for ( k = 0; k < i; k++ ) {
                d2[i][k] = 0.0;
                for ( j = 0; j < 3; j++ )
                    d2[i][k] += ( particles[i].position[j] - particles[k].position[j] ) *
                        ( particles[i].position[j] - particles[k].position[j] );
                d2[k][i] = d2[i][k];
            }
    last_time = present_time;
    glutPostRedisplay();
}
```

# Example (Cont'd)

```
void collision( int n )
/* tests for collisions against cube and reflect particles if necessary */
{
    int i;
    for ( i = 0; i < 3; i++ ) {
        if ( particles[n].position[i] >= 1.0 ) {
            particles[n].velocity[i] = -coef * particles[n].velocity[i];
            particles[n].position[i] = 1.0 - coef * ( particles[n].position[i] - 1.0 );
        }
        if ( particles[n].position[i] <= -1.0 ) {
            particles[n].velocity[i] = -coef * particles[n].velocity[i];
            particles[n].position[i] = -1.0 - coef * ( particles[n].position[i] + 1.0 );
        }
    }
}
```

# Example (Cont'd)

```
void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT );
    for ( i = 0; i < num_particles; i++ ) {
        point_colors[i + 24] = colors[particles[i].color];
        points[i + 24] = particles[i].position;
    }
    glBufferSubData( GL_ARRAY_BUFFER, 0, sizeof(points), points );
    glBufferSubData( GL_ARRAY_BUFFER, sizeof(points), sizeof(point_colors),
point_colors );
    glDrawArrays( GL_POINTS, 24, num_particles );
   glutSwapBuffers();
}
```

# Reading Assignments

**Chapter 9.4 – 9.9 in Angel & Shreiner**

**Chapter 10 & 11 in Angel & Shreiner**

**Chapter 9 – 12 in Shreiner et al.**