물리기반 모델링

충돌

동명대학교
강영민

충격(impulse)

- 충격량
  - 아주 짧은 시간에 작용하는 힘
  - 예
    - 총에서 발사되는 총알에 작용하는 힘
    - 충돌하는 물체들 사이에 작용하는 힘
- 물리적 의미
  - 충격량: 운동량의 변화량과 같은 벡터양
    - 선형 충격량
      - = $m(\mathbf{v}_+ - \mathbf{v}_-)$
    - 회전 충격량
      - = $\mathbf{I}(\omega_+ - \omega_-)$

충격량의 계산

- 총에서 발사된 총알
  - 총알의 질량: 0.15 kg
  - 총구에서의 총알 속도: 756 m/s
  - 총신의 길이: 0.610 m
  - 총알이 총신을 통과하는 데에 걸리는 시간: 0.0008 s
- 충격량 = 운동량의 변화
  - mv = 0.15*756 kg m/s = 113.4 kgm/s
- 평균 충격량 힘 = 충격량 / 시간
  - 113.4 / 0.00008 N = 141,750 N

운동량 보존

- 두 개의 객체 (각각의 질량은 m1과 m2) 충돌
  - 충돌 이전의 속도 v-
  - 충돌 이후의 속도 v+

$$m_1\mathbf{v}_1^+ + m_2\mathbf{v}_2^+ = m_1\mathbf{v}_1^- + m_2\mathbf{v}_2^-$$

운동 에너지의 보존

- 선형 운동 에너지

$$K_l = \frac{1}{2} m |\mathbf{v}|^2$$

- 회전 운동 에너지

$$K_a = \frac{1}{2} I |\omega|^2$$

- 운동 에너지가 보존된다면 다음이 만족됨

$$m_1 {\mathbf{v}_1^{+}}^2 + m_2 {\mathbf{v}_2^{+}}^2 = m_1 {\mathbf{v}_1^{-}}^2 + m_2 {\mathbf{v}_2^{-}}^2$$

충돌 객체들의 속도 변화

- 운동량 보존

$$m_1\mathbf{v}_1^+ + m_2\mathbf{v}_2^+ = m_1\mathbf{v}_1^- + m_2\mathbf{v}_2^-$$

$$\boxed{m_1(\mathbf{v}_1^+ - \mathbf{v}_1^-) = -m_2(\mathbf{v}_2^+ - \mathbf{v}_2^-)}$$

- 에너지 보존

$$m_1{\mathbf{v}_1^+}^2 + m_2{\mathbf{v}_2^+}^2 = m_1{\mathbf{v}_1^-}^2 + m_2{\mathbf{v}_2^-}^2$$

$$m_1({\mathbf{v}_1^+}^2 - {\mathbf{v}_1^-}^2) = -m_2({\mathbf{v}_2^+}^2 - {\mathbf{v}_2^-}^2)$$

$$m_1(\mathbf{v}_1^+ - \mathbf{v}_1^-)(\mathbf{v}_1^+ + \mathbf{v}_1^-) = -m_2(\mathbf{v}_2^+ - \mathbf{v}_2^-)(\mathbf{v}_2^+ + \mathbf{v}_2^-)$$

$$\boxed{\mathbf{v}_1^+ + \mathbf{v}_1^- = \mathbf{v}_2^+ + \mathbf{v}_2^-}$$

충돌 이후의 속도 구하기

$$m_1(\mathbf{v}_1^+ - \mathbf{v}_1^-) = -m_2(\mathbf{v}_2^+ - \mathbf{v}_2^-)$$

$$\mathbf{v}_1^+ + \mathbf{v}_1^- = \mathbf{v}_2^+ + \mathbf{v}_2^-$$

$$m_1\mathbf{v}_1^+ + m_2\mathbf{v}_2^+ = m_1\mathbf{v}_1^- + m_2\mathbf{v}_1^-$$

$$\mathbf{v}_1^+ - \mathbf{v}_2^+ = \mathbf{v}_2^- - \mathbf{v}_1^-$$

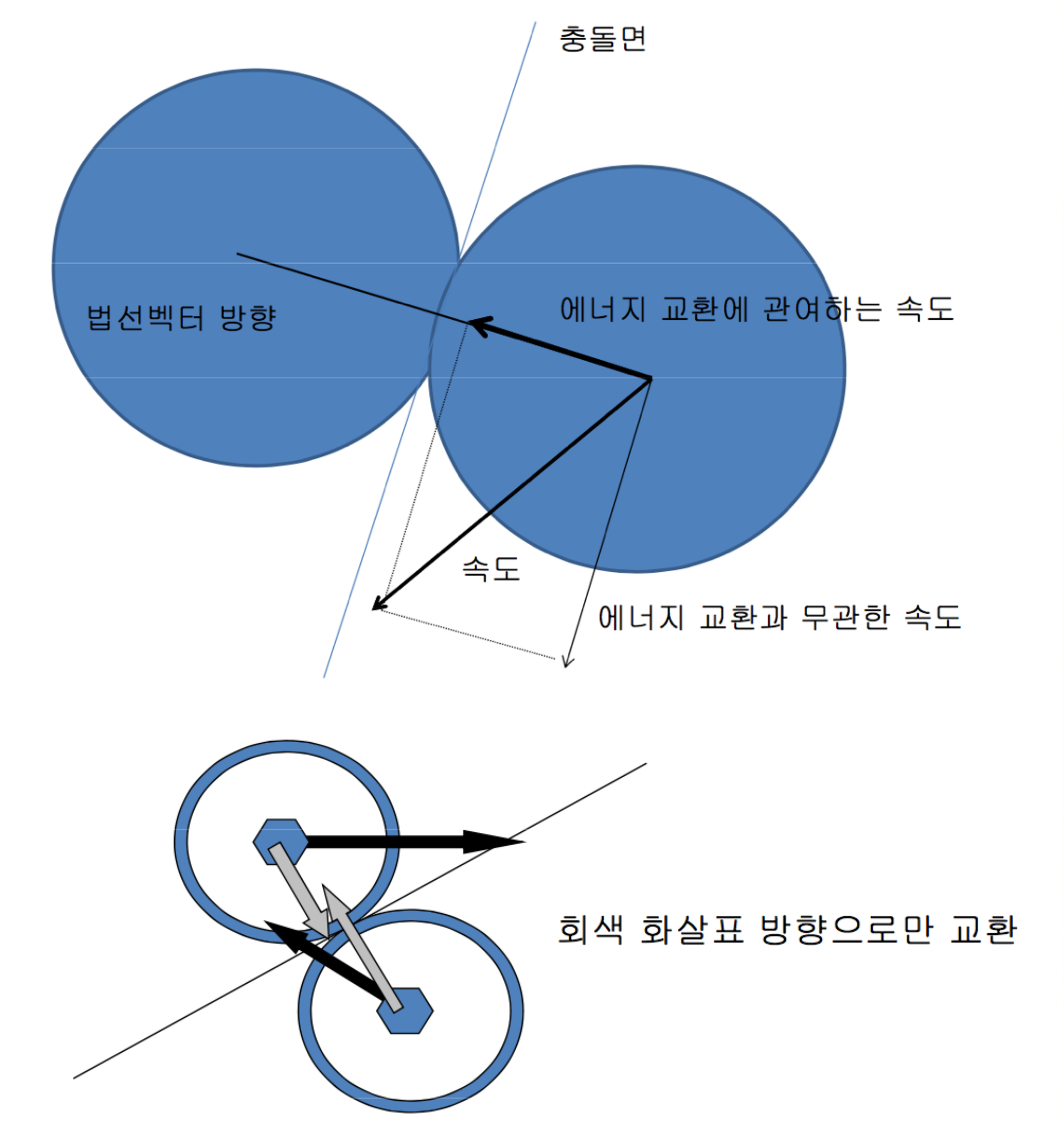
$$\mathbf{v}_1^+ = \frac{(m_1 - m_2)\mathbf{v}_1^- + 2m_2\mathbf{v}_2^-}{m_1 + m_2}$$

$$\mathbf{v}_2^+ = \frac{2m_1\mathbf{v}_1^- + (m_2 - m1)\mathbf{v}_2^-}{m_1 + m_2}$$

속도의 변화

- 정면 충돌과 빗겨 맞는 충돌
    - 정면 충돌
        - 앞의 수식을 그대로 적용
    - 빗겨 맞는 충돌
        - 충돌 작용선을 알아야 함
        - 이 충돌 작용선으로 작용하는 속도 성분만 변경됨

속도의 변화



회색 화살표 방향으로만 교환

충돌 작용선

- 두 입자의 중심점
  - p1, p2
- 충돌 작용선의 방향
  - N = (p1-p2) / | p1-p2|
- 충돌의 감지
  - 입자의 반지름을 이용
    - r1, r2
    - |p1-p2| < r1+r2

충돌 작용선 방향의 속도

- 충돌 이전에 이 충돌 작용선 방향의 속도 크기

$$v_1^{-} = \mathbf{v}_1 \cdot \mathbf{N}$$

$$v_2^{-} = \mathbf{v}_2 \cdot \mathbf{N}$$

- 충돌 처리
  - 두 입자가 서로 접근할 때에만 처리 (감지는 거리로, 처리는 여부는 속도로)

$$v_2^{-} - v_1^{-} > 0$$

속도의 갱신

$$v_1^+ = \frac{(m_1 - m_2)v_1^- + 2m_2 v_2^-}{m_1 + m_2}$$

$$v_2^+ = \frac{(m_2 - m_1)v_2^- + 2m_1 v_1^-}{m_1 + m_2}$$

$$\boxed{\begin{aligned} \mathbf{v}_1 &= \mathbf{v}_1 - v_1^- \mathbf{N} + v_1^+ \mathbf{N} \\ \mathbf{v}_2 &= \mathbf{v}_2 - v_2^- \mathbf{N} + v_2^+ \mathbf{N} \end{aligned}}$$

물리기반 모델링

비탄성 충돌

동명대학교
강영민

충격량과 탄성 계수

- 충격량(impulse)
  - J
    - 운동량의 변화
    - J = m(v₊ - v₋)
- 탄성계수
  - $$\epsilon = \frac{-(v_{1+} - v_{2+})}{v_{1-} - v_{2-}}$$

충격량 J를 알면

- $v_+ = J / m + v_-$

충격량과 탄성의 관계

- 3개의 식이 필요

$$|\mathbf{J}| = m_1(v_{1+} - v_{1-})$$
$$-|\mathbf{J}| = m_1(v_{2+} - v_{2-})$$

$$\epsilon = \frac{-(v_{1+} - v_{2+})}{v_{1-} - v_{2-}}$$

⇨

$$v_{1+} = |\mathbf{J}|/m_1 + v_{1-}$$
$$v_{2+} = -|\mathbf{J}|/m_1 + v_{2-}$$

$$\epsilon = \frac{-(v_{1+} - v_{2+})}{v_{1-} - v_{2-}}$$

$$\epsilon(v_{1-} - v_{2-}) = -(v_{1+} - v_{2+})$$

충격량

- 충격량과 충돌이전 속도의 관계

$$\epsilon(v_{1-} - v_{2-}) = -(|\mathbf{J}|/m_1 + v_{1-} + |\mathbf{J}|/m_2 - v_{2-})$$
$$\epsilon(v_{1-} - v_{2-}) = -(|\mathbf{J}|(1/m_1 + 1/m_2) + v_{1-} - v_{2-})$$

- 충격량의 크기

$$|\mathbf{J}| = (1+\epsilon)(v_{1-} - v_{2-})/(1/m_1 + 1/m_2)$$

속도의 갱신

- 충돌한 방향으로 속도의 갱신

$$v_{1+} = v_{1-} + |\mathbf{J}|/m_1$$

$$v_{2+} = v_{2-} - |\mathbf{J}|/m_2$$

충돌처리

```
void CDynamicSimulator::collisionHandler(int i, int j) {
    // collision detect
    CVec3d p1; p1 = particle[i].getPosition();
    CVec3d p2; p2 = particle[j].getPosition();
    CVec3d N ; N = p1 - p2;
    double dist = N.len();
    double e = 0.1;
    double penetration = particle[i].getRadius() + particle[j].getRadius() - dist;

    if(penetration>0) {

     // collision detected
     N.normalize();
     CVec3d v1; v1 = particle[i].getVelocity();
     CVec3d v2; v2 = particle[j].getVelocity();
     double v1N = v1 ^ N; // velocity along the line of action ( dot product of v1 and N )
     double v2N = v2 ^ N; // velocity along the line of action ( dot product of v2 and N )
     double m1 = particle[i].getMass();
     double m2 = particle[j].getMass();
     // approaching ?
     if( v1N-v2N < 0 ) { // approaching
          double vr = v1N - v2N;
          double J = -vr*(e+1.0)/(1.0/m1 + 1.0/m2);
          double v1New = v1N + J/m1;
          double v2New = v2N - J/m2;
          v1 = v1 - v1N * N + v1New*N;
          v2 = v2 - v2N * N + v2New*N;
          particle[i].setVelocity(v1.x, v1.y, v1.z);
          particle[j].setVelocity(v2.x, v2.y, v2.z);
     }
     p1 = p1 + (0.5*(1.0+e)*penetration)*N;
     p2 = p2 - (0.5*(1.0+e)*penetration)*N;
     particle[i].setPosition(p1.x, p1.y, p1.z);
     particle[j].setPosition(p2.x, p2.y, p2.z);
    }
}
```

다수의 입자 만유인력

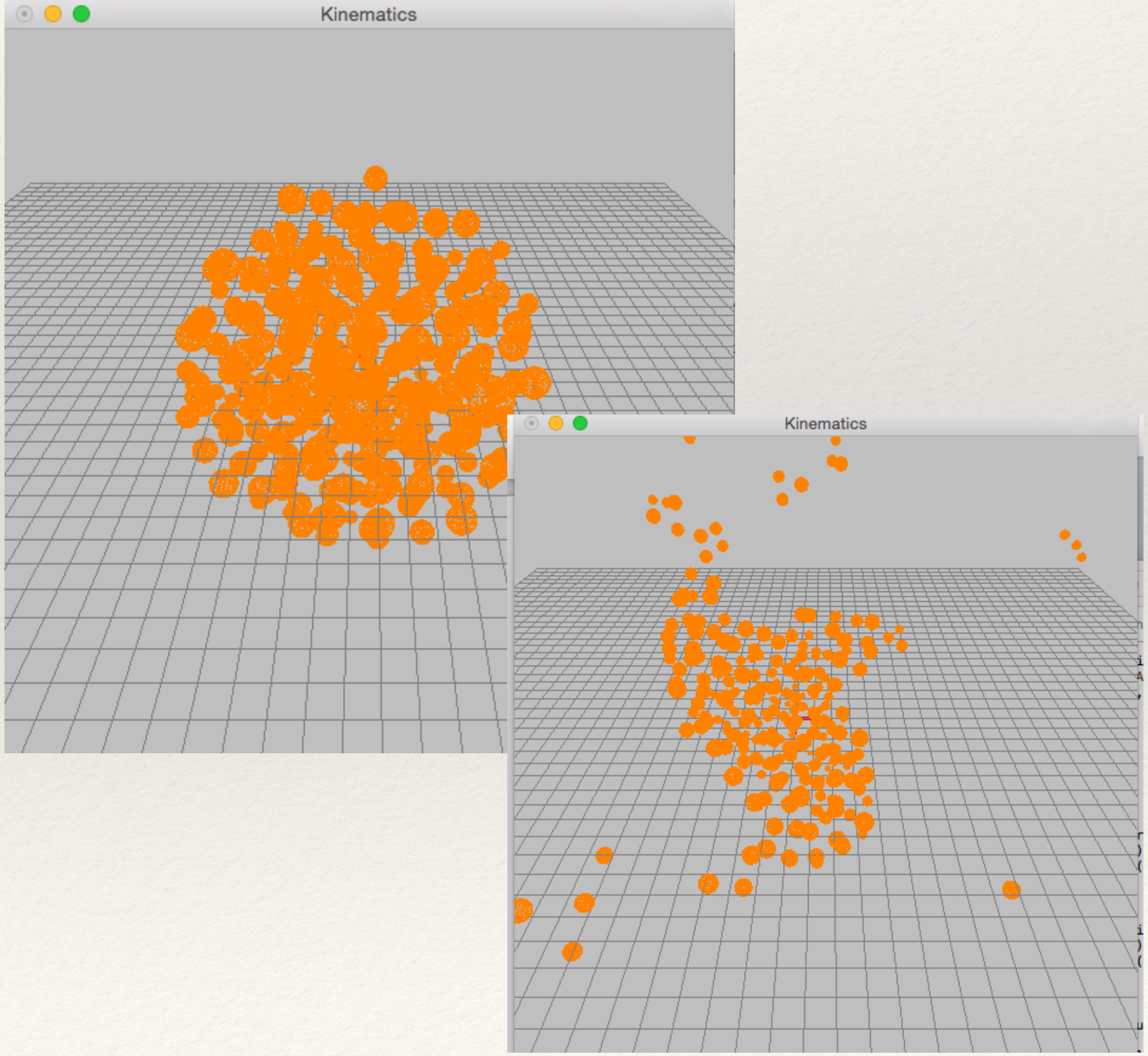
- 랜덤하게 입자를 생성
- 입자간 인력 작용
- 인력의 크기
  - $\dfrac{m_i m_j}{r^2}$ 에 비례

인력 계산

```cpp
CVec3d CDynamicSimulator::computeAttraction(int i, int j) {
    // collision detect
    CVec3d xi; xi = particle[i].getPosition();
    CVec3d xj; xj = particle[j].getPosition();
    CVec3d xij; xij = xj-xi;
    double dist = xij.len();
    xij.normalize();
    double mi = particle[i].getMass();
    double mj = particle[j].getMass();

    double G = 5.5;
    CVec3d force;
    force = (G*mi*mj/(dist*dist))*xij;
    return force;

}
```

시뮬레이션

```
void CDynamicSimulator::doSimulation(double dt, double currentTime) {

    if(dt>0.01)dt=0.01; // maximum dt

    CVec3d forcei;
    CVec3d forcej;
    for (int i=0; i<NUMPARTS; i++) {
        for (int j=i+1; j<NUMPARTS; j++) {
            forcei = computeAttraction(i, j);
            forcej = -1.0*forcei;
            particle[i].addForce(forcei);
            particle[j].addForce(forcej);
        }
    }

    for (int i=0; i<NUMPARTS; i++) {
        particle[i].simulate(dt, currentTime);
    }

    for (int i=0; i<NUMPARTS; i++) {
        for (int j=i+1; j<NUMPARTS; j++) {
            collisionHandler(i, j);
        }
    }

}
```



물리기반 모델링

스프링과 댐퍼

동명대학교
강영민

스프링 댐퍼 (Spring and damper)

- 스프링 댐퍼 모델
    - 두 입자의 상호작용
        - 두 입자를 연결하는 스프링의 힘
            - 스프링 힘
        - 스프링 운동에 의한 에너저 소산
            - 댐핑 힘

스프링 힘

- 스프링 힘
  - 후크(Hooke)의 법칙
    - 스프링에 작용하는 힘의 크기
      - 변형된 길이에 비례 $l - l_0$
      - 스프링 상수에 비례 (스프링의 고유한 특성) : k_s
    - 스프링 힘의 방향
      - 스프링 방향

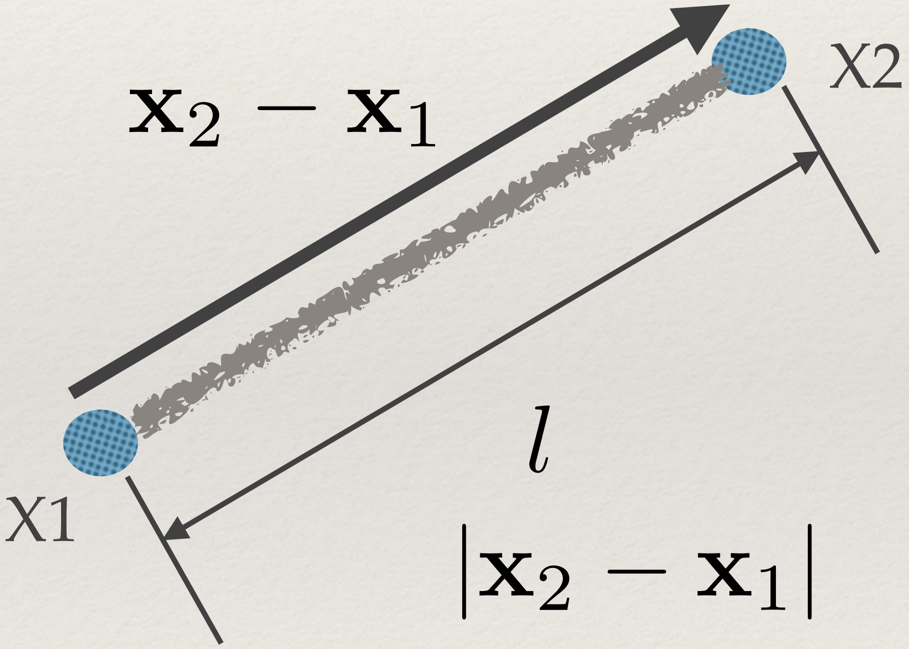
스프링 힘의 계산

- 힘의 크기

$$|\mathbf{f}_s| = k_s|l - l_0|$$

- 힘의 방향

$$\frac{\mathbf{x}_2 - \mathbf{x}_1}{|\mathbf{x}_2 - \mathbf{x}_1|}$$


$\mathbf{x}_2 - \mathbf{x}_1$
X2
X1
l
$|\mathbf{x}_2 - \mathbf{x}_1|$

스프링 힘

- 계산 방법

$$\mathbf{f}_s^i = k_{ij}(l - l_0)\frac{\mathbf{x}_j - \mathbf{x}_i}{|\mathbf{x}_j - \mathbf{x}_i|}$$

$$\mathbf{f}_s^j = -\mathbf{f}_s^i$$

댐핑

- 스프링 진동은 서서히 멈춘다
  - 에너지를 잃게 만들어야 함
- 간단한 댐핑
  - 속도의 반대 방향으로 감속

$$\mathbf{f}_d^i = -k_d \mathbf{v}^i$$

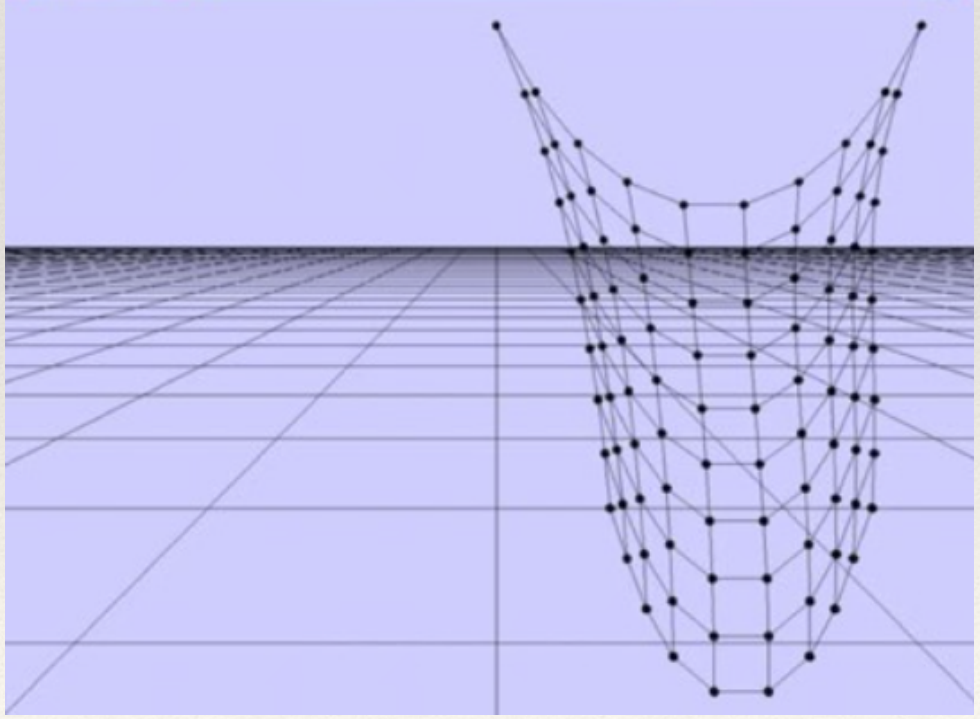
- 문제점
  - 스프링에 의해 소실되는 에너지가 아니라 공기저항 같은 효과
- 개선방법
  - 연결된 두 입자의 상대속도에 댐핑 적용
    - 입자가 현재 상태를 바꾸려고 하는 운동에 대해서 저항

$$\mathbf{f}_d^i = k_d(\mathbf{v}^j - \mathbf{v}^i)$$

최종 모델

$$\mathbf{f}_{ij}^{i} = k_{ij}(l - l_0)\frac{\mathbf{x}_j - \mathbf{x}_i}{|\mathbf{x}_j - \mathbf{x}_i|} + k_d(\mathbf{v}_j - \mathbf{v}_i)$$

$$\mathbf{f}_{ij}^{j} = -\mathbf{f}_{ij}^{i}$$



물리기반 모델링

당구게임

동명대학교
강영민

Particle.h - 당구공

```cpp
enum DrawMode {
    POINT_DRAW,
    SPHERE_DRAW
};

class CParticle {
public:
   int type;
   double radius;
   double mass;

   CVec3d loc, vel, force, gravity;
   CVec3d color;

private:
    void forceIntegration(double dt, double et);

public:
   CParticle();
```

```cpp
   void setPosition(double x, double y, double z);
   void setVelocity(double vx, double vy, double vz);
   void setMass(double m);
   void setRadius(double r);
   void setColor(double r, double g, double b);

   CVec3d getPosition();
   CVec3d getVelocity();
   double getMass();
   double getRadius();


   void resetForce(void);
   void addForce(CVec3d &f);

   void drawWithGL(int drawMode = SPHERE_DRAW);
   void simulate(double dt, double eT);
};
```

Particle.cpp - 당구공

```
CParticle::CParticle() {
    radius = 1.0f;
    loc.set(0.0, 0.0, 0.0);
}
void CParticle::setPosition(double x, double y, double z) {
    loc.set(x,y,z);
}
void CParticle::setVelocity(double vx, double vy, double vz) {
   vel.set(vx,vy,vz);
}
void CParticle::setMass  (double m) { mass = m; }
void CParticle::setRadius(double r) { radius = r; }
void CParticle::setColor (double r, double g, double b) { color.set(r,g,b); }

CVec3d CParticle::getPosition() { return loc ; }
CVec3d CParticle::getVelocity() { return vel ; }
double CParticle::getMass()     { return mass; }
double CParticle::getRadius()   { return radius; }
```

Particle.cpp - 당구공

```
void CParticle::drawWithGL(int drawMode) {
    glColor3f(color.x, color.y, color.z);

    glPushMatrix();
    glTranslated(loc[0], loc[1], loc[2]);
    if (drawMode == SPHERE_DRAW) {
        glutWireSphere(radius, 30, 30);
    }
    else {
        glBegin(GL_POINTS);
        glVertex3f(0,0,0);
        glEnd();
    }
    glPopMatrix();
}
void CParticle::forceIntegration(double dt, double et) {
    if(dt>0.1) dt=0.1;
    vel = vel + dt*((1.0/mass) * force );
    loc = loc + dt*vel;
}
void CParticle::simulate(double dt, double et) {
    forceIntegration(dt, et);
    if(this->vel.len()<10) vel.set(0.0,0.0,0.0);
}
void CParticle::resetForce(void) {  this->force.set(0.0, 0.0, 0.0); }
void CParticle::addForce(CVec3d &f) { this->force = this->force + f; }
```

main.cpp - Game Control

```cpp
void key_ready(unsigned char key) {
    switch (key) {
    case 's': // start game
            Simulator->start(); myWatch.start();
            ((CDynamicSimulator *)Simulator)->setMode(AIMING);
            break;
    }
}
void key_aiming(unsigned char key) {
    switch (key) {
        case '.': ((CDynamicSimulator *)Simulator)->rotateAim( 0.05);break;
        case ',': ((CDynamicSimulator *)Simulator)->rotateAim(-0.05);break;
        case 'm': ((CDynamicSimulator *)Simulator)->rotateAim(-0.01);break;
        case '/': ((CDynamicSimulator *)Simulator)->rotateAim( 0.01);break;
        case ' ': ((CDynamicSimulator *)Simulator)->shot();break;
    }
}
void key_simulating(unsigned char key) {
    switch (key) {
        case 'p': myWatch.pause(); Simulator->pause(); break;
        case 'r': myWatch.resume(); break;
        case ' ': ((CDynamicSimulator *)Simulator)->turnOver();break;
        default: break;
    }
}

void KEY_turnover(unsigned char key) {
    switch (key) {
        case ' ':((CDynamicSimulator *)Simulator)->setMode(AIMING); break;
    }
}
void keyboardFunction(unsigned char key, int x, int y) {
    if (key == 27) exit(0);
    gameMode mode = ((CDynamicSimulator *)Simulator)->getMode();
    switch(mode) {
    case READY: key_ready(key); break;
    case AIMING: key_aiming(key); break;
    case SIMULATING: key_simulating(key); break;
    case TURNOVER: KEY_turnover(key); break;
    default: break;
    }
}
```

main.cpp - Display/Idle function

```cpp
void displayFunction(void) {


   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   setupCamera(0, 2500, 0, 0, 0, 0, 1, 0, 0);


   // check DT (in microsecond) from StopWatch and store it to "deltaTime" (in seconds)
   deltaTime = myWatch.checkAndComputeDT() / 1000000.0;
   currentTime = myWatch.getTotalElapsedTime() / 1000000.0;

   Simulator->actions(deltaTime, currentTime);
   // actions < doBeforeSimulation, doSimulation, doAfterSimulation >

   glutSwapBuffers();
}
```

DynamicSimulator.h

```
#include "Simulator.h"
#include "Particle.h"

#define NUMBALLS 4
#define TABLE_W 1420
#define TABLE_H 2840
#define BALL_RADIUS 40.75

typedef enum MODE {
    READY,
    AIMING,
    SIMULATING,
    TURNOVER
} gameMode;

enum TURNS {
    PLAYER1,
    PLAYER2,
    NUMPLAYERS
};
```

Definitions and enumerations

DynamicSimulator.h - 클래스

```
class CDynamicSimulator : public CSimulator {
    CParticle balls[NUMBALLS];
    TURNS turn;
    MODE mode;
    CVec3d aim;
    float aimAngle;
public:
    CDynamicSimulator();
    void init(void);
    void clean(void);
    MODE getMode(void);
    void setMode(MODE m);
    void rotateAim(double angle);
    void shot(void);
    void turnOver(void);
private:
    void doBeforeSimulation(double dt, double currentTime);
    void doSimulation(double dt, double currentTime);
    void doAfterSimulation(double dt, double currentTime);
    void visualize(void);

    CVec3d computeAttraction(int i, int j);
   void collisionHandler(int i, int j);
    void floorDrag(void);
    void cushion(void);
};
```

DynamicSimulator.cpp

```
void CDynamicSimulator::init() {

    turn = PLAYER1;
    mode = READY;

    for(int i=0;i<NUMBALLS;i++) {
        balls[i].setRadius(BALL_RADIUS);
        balls[i].setMass(0.16);
        balls[i].setVelocity(0.0, 0.0, 0.0);
    }
    balls[0].setPosition( TABLE_W/20.0, BALL_RADIUS, 3.0*TABLE_H/8.0);
    balls[0].setColor(1.0, 1.0, 1.0);
    balls[1].setPosition(-TABLE_W/20.0, BALL_RADIUS, 3.0*TABLE_H/8.0);
    balls[1].setColor(1.0, 0.0, 0.0);
    balls[2].setPosition( 0, BALL_RADIUS,-3.0*TABLE_H/8.0);
    balls[2].setColor(1.0, 1.0, 0.0);
    balls[3].setPosition( 0, BALL_RADIUS,-2.0*TABLE_H/8.0);
    balls[3].setColor(1.0, 0.0, 0.0);

    aim.set(1.0, 0.0, 0.0);
}
```

DynamicSimulator.cpp

```
void CDynamicSimulator::doSimulation(double dt, double currentTime) {

    if(dt>0.01)dt=0.01; // maximum dt

    if(mode!=SIMULATING) return;

    floorDrag();

    for (int i=0; i<NUMBALLS; i++) {
        balls[i].simulate(dt, currentTime);
    }

    cushion();


    for (int i=0; i<NUMBALLS; i++) {
        for (int j=i+1; j<NUMBALLS; j++) {
            collisionHandler(i, j);
        }
    }
}

void CDynamicSimulator::doAfterSimulation(double dt, double currentTime) {
    for(int i=0;i<NUMBALLS;i++) {
    balls[i].resetForce();
   }
}
```

DynamicSimulator.cpp

```
void CDynamicSimulator::visualize(void) {
    // Draw Table
    glColor3f(0.0, 0.5, 0.0);
    glBegin(GL_QUADS);
    glVertex3f(-TABLE_W/2.0, 0.0,-TABLE_H/2.0);
    glVertex3f(-TABLE_W/2.0, 0.0, TABLE_H/2.0);
    glVertex3f( TABLE_W/2.0, 0.0, TABLE_H/2.0);
    glVertex3f( TABLE_W/2.0, 0.0,-TABLE_H/2.0);
    glEnd();

    for(int i=0;i<NUMBALLS;i++) {
        balls[i].drawWithGL(SPHERE_DRAW);
    }

    if (mode == AIMING) {
        CVec3d pos; pos = balls[turn*2].getPosition();
        glBegin(GL_LINES);
        glVertex3f(pos.x, pos.y, pos.z);
        glVertex3f(pos.x+aim.x*2000.0, pos.y+aim.y*2000.0, pos.z+aim.z*2000.0);
        glEnd();
    }
}
```

DynamicSimulator.cpp

```cpp
void CDynamicSimulator::collisionHandler(int i, int j) {
   // collision detect
    CVec3d p1; p1 = balls[i].getPosition();
    CVec3d p2; p2 = balls[j].getPosition();
    CVec3d N ; N = p1 - p2;
   double dist = N.len();
    double e = 0.9;
   if(dist < balls[i].getRadius() + balls[j].getRadius()) {
     double penetration = balls[i].getRadius() + balls[j].getRadius() - dist;
    // collision detected
    N.normalize();
    CVec3d v1; v1 = balls[i].getVelocity();
    CVec3d v2; v2 = balls[j].getVelocity();
    double v1N = v1 ^ N; // velocity along the line of action
    double v2N = v2 ^ N; // velocity along the line of action
    double m1 = balls[i].getMass();
    double m2 = balls[j].getMass();
    // approaching ?
    if( v1N-v2N < 0 ) { // approaching
        double vr = v1N - v2N;
        double J = -vr*(e+1.0)/(1.0/m1 + 1.0/m2);
        double v1New = v1N + J/m1;
        double v2New = v2N - J/m2;
        v1 = v1 - v1N * N + v1New*N;
        v2 = v2 - v2N * N + v2New*N;
        balls[i].setVelocity(v1.x, v1.y, v1.z);
        balls[j].setVelocity(v2.x, v2.y, v2.z);
    }
     p1 = p1 + 0.5*((1.0+e)*penetration)*N;
     p2 = p2 - 0.5*((1.0+e)*penetration)*N;
     balls[i].setPosition(p1.x, p1.y, p1.z);
     balls[j].setPosition(p2.x, p2.y, p2.z);
   }
}
```

DynamicSimulator.cpp

```
void CDynamicSimulator::floorDrag(void) {
    CVec3d vel, dragForce;
    double drag = 0.05;
    for(int i=0;i<NUMBALLS;i++) {
        vel = balls[i].getVelocity();
        dragForce = -drag*vel;
        balls[i].addForce(dragForce);
    }
}
MODE CDynamicSimulator::getMode(void)   { return mode; }
void CDynamicSimulator::setMode(MODE m) { mode = m; }

void CDynamicSimulator::rotateAim(double angle) {
    aimAngle+=angle;
    if(aimAngle>3.141592*2.0) aimAngle-=3.141592*2.0;
    aim.set(cos(aimAngle), 0.0, sin(aimAngle));
}
void CDynamicSimulator::shot(void) {
    balls[turn*2].setVelocity(5000*aim.x, 0.0, 5000*aim.z);
    mode = SIMULATING;
}
void CDynamicSimulator::turnOver(void) {
    for(int i=0;i<NUMBALLS;i++) balls[i].setVelocity(0.0, 0.0, 0.0);
    turn = turn==PLAYER1?PLAYER2:PLAYER1;
    mode = TURNOVER;
}
```

DynamicSimulator.cpp

```cpp
void CDynamicSimulator::cushion(void) {
    // collision detect
    for(int i=0;i<NUMBALLS; i++) {
        CVec3d pos; pos = balls[i].getPosition();
        CVec3d vel; vel = balls[i].getVelocity();
        CVec3d N;
        double r = balls[i].getRadius();
        double pene = 0.0;
        if(pos.x + r > TABLE_W/2.0) {
            pene = pos.x + r - TABLE_W/2.0;
            N.set(-1.0, 0, 0);
        }
        else if(pos.x - r < -TABLE_W/2.0) {
            pene = -TABLE_W/2.0 - pos.x + r;
            N.set(1.0,0.0,0.0);
        }
        else if(pos.z + r > TABLE_H/2.0) {
            pene = pos.z + r - TABLE_H/2.0;
            N.set(0.0, 0.0, -1.0);
        }
        else if(pos.z - r < -TABLE_H/2.0) {
            pene = -TABLE_H/2.0 - pos.z + r;
            N.set(0.0, 0.0, 1.0);
        }
        double vN = vel^N;
        if (vN<0.0) { // penetrating
            vel = vel - (2.0 * vN)*N;
        }
        pos = pos + (2.0*pene)*N;
        balls[i].setVelocity(vel.x, vel.y, vel.z);
        balls[i].setPosition(pos.x, pos.y, pos.z);
    }
}
```

Result