물리기반 모델링 - *Part 1*

물리기반 모델링의 기초

동명대학교 게임공학과
강영민

물리기반 모델링

1.1 질량, 힘, 시간

동명대학교 게임공학과
강영민

물리기반 모델링은 왜 하나?

- 애니메이션/시뮬레이션
  - 사실적인 움직임이 필요
- 애니메이션
  - 시간에 따른 상태의 변화
  - 상태 - 위치, 속도...
    - 물리의 중요한 연구 대상
- 사실적인 움직임
  - 물리에 의해 결정됨

운동에 대한 연구

- 뉴턴
  - 고전 물리에서 운동에 대한 이론을 정립
  - 프린키피아 (*Philosophiae Naturalis Principia Mathematica*)
- 뉴턴의 운동법칙
  - 1법칙: 일정한 운동을 하는 객체는 외부의 힘이 가해지기 전에는 그 운동을 유지하려고 한다. (관성)
  - 2법칙: $\mathbf{F}=m\mathbf{a}$.
  - 3법칙: 모든 작용에는 같은 크기의 반대방향으로 반작용이 있다.

질량

- 질량
  - 힘에 의한 가속에 저항하는 속성
  - 누적된 밀도 (kg/m^3)
    - 질량 = 밀도의 적분

$$m = \int \rho dV$$

질량 중심

- 이론

$$x_c = \frac{\int x_0 dm}{m}$$

$$y_c = \frac{\int y_0 dm}{m}$$

$$z_c = \frac{\int z_0 dm}{m}$$

$$c_g = \frac{\sum (c_{g_i} m_i)}{m}$$

- 계산

$$\boxed{\begin{aligned} x_c &= \frac{\sum x_i m_i}{\sum m_i} \\ y_c &= \frac{\sum y_i m_i}{\sum m_i} \\ z_c &= \frac{\sum z_i m_i}{\sum m_i} \end{aligned}}$$

V

dm at (x_0,y_0,z_0)

뉴턴의 운동 제2법칙

- 힘 = 질량 x 가속

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

- 총 힘의 합 $\mathbf{f}_{net}$
  - 가해진 모든 힘의 합 $\mathbf{f}_{net} = \sum \mathbf{f}_i$

$$\sum f_x = ma_x$$
$$\sum f_y = ma_y$$
$$\sum f_z = ma_z$$

- 가속은 힘의 총합에 의해 결정된다

$$\mathbf{a} = \frac{\mathbf{f}_{net}}{m}$$

선운동량과 미분

- 선운동량: **G**
  - 질량 x 속도
  - $\mathbf{G} = m\mathbf{v}$
- 선운동량을 시간에 대해 미분하면

$$\frac{d\mathbf{G}}{dt} = \frac{dm\mathbf{v}}{dt} = m\frac{d\mathbf{v}}{dt} = m\mathbf{a}$$

  - 결과는 힘

$$\frac{d\mathbf{G}}{dt} = \sum \mathbf{f}$$

시간

- 고전 물리
  - 시간은 불변
- 현대 물리
  - 시간은 가변적
  - 빛의 속도는 상수: c
    - $c = 299{,}792{,}458\ m/s$

상대론적 시간

수업 시간에 사용하지는 않을 것이지만 재미로 다뤄보는 상대론적 시간

$$c = 2L/\Delta t$$

속도 v로 이동하는 우주선에서 빛의 이동

우주선에서

$$c = 2L/\Delta t_s$$

지구에서

빛의 속도 c 가 상수라면
두 공간의 시간은 서로 달라야!

$$c = 2H/\Delta t_e$$

시간 확장

- 간단한 기하

$$H^2 = L^2 + \left(\frac{\mathbf{v}\Delta t_e}{2}\right)^2 \Rightarrow \left(\frac{c\Delta t_e}{2}\right)^2 = L^2 + \left(\frac{\mathbf{v}\Delta t_e}{2}\right)^2$$

- 시간 확장의 계산

$$\boxed{\begin{aligned} 4L^2 &= c^2\Delta t_e^2 - \mathbf{v}^2\Delta t_e^2 \\ 4L^2 &= (c^2 - \mathbf{v}^2)\Delta t_e^2 \\ \frac{4L^2}{c^2} &= (1 - \frac{\mathbf{v}^2}{c^2})\Delta t_e^2 \end{aligned}} \Rightarrow \boxed{\begin{aligned} \frac{2L}{c} &= \sqrt{1 - \frac{\mathbf{v}^2}{c^2}}\Delta t_e \\ \Delta t &= \sqrt{1 - \frac{\mathbf{v}^2}{c^2}}\Delta t_e \end{aligned}}$$

$$\Delta t_e = \frac{1}{\sqrt{1 - \frac{\mathbf{v}^2}{c^2}}}\Delta t$$

시간과 애니메이션

- 애니메이션
  - 시간에 따른 변화
- 컴퓨터 애니메이션
  - 시간에 대해 적절한 물리적 상태를 계산하는 것
- 시간을 측정해야만 애니메이션이 가능
  - 시간 측정 프로그램이 필요

Stop Watch (header)

```
#ifndef _STOPWATCH_YMKANG_H
#define _STOPWATCH_YMKANG_H

#ifdef WIN32   // Windows system specific
#include <windows.h>
#else          // Unix based system specific
#include <sys/time.h>
#endif

class StopWatch {
#ifdef WIN32
    LARGE_INTEGER frequency;                    // ticks per second
    LARGE_INTEGER startCount;                   //
    LARGE_INTEGER endCount;                     //
#else
    timeval startCount;                         //
    timeval endCount;                           //
#endif
    double startTimeInMicroSec;
    double endTimeInMicroSec;
public:
    StopWatch();
    void start();                // start StopWatch and record time to "startCount"
    void stop();                // stop StopWatch and record time to "endCount"
    double getElapsedTime();    // return the elapsed time at the last stop since the last start (microsec)
};

#endif
```

Stop Watch (implementation)

```
/*
 *  StopWatch.cpp
 *  Young-Min Kang
 *  Tongmyong University
 *
 */

#include "StopWatch.h"

StopWatch::StopWatch() {
#ifdef WIN32
    QueryPerformanceFrequency(&frequency);
    startCount.QuadPart = 0;
    endCount.QuadPart = 0;
#else
    startCount.tv_sec = startCount.tv_usec = 0;
    endCount.tv_sec = endCount.tv_usec = 0;
#endif
    startTimeInMicroSec = endTimeInMicroSec = 0.0;
}

void StopWatch::start() {
#ifdef WIN32
    QueryPerformanceCounter(&startCount);
#else
    gettimeofday(&startCount, NULL);
#endif
}

void StopWatch::stop() {
#ifdef WIN32
    QueryPerformanceCounter(&endCount);
#else
    gettimeofday(&endCount, NULL);
#endif
}

double StopWatch::getElapsedTime(){
#ifdef WIN32
    startTimeInMicroSec = startCount.QuadPart * (1000000.0 / frequency.QuadPart);
    endTimeInMicroSec = endCount.QuadPart * (1000000.0 / frequency.QuadPart);
#else
    startTimeInMicroSec = (startCount.tv_sec * 1000000.0) + startCount.tv_usec;
    endTimeInMicroSec = (endCount.tv_sec * 1000000.0) + endCount.tv_usec;
#endif
    return endTimeInMicroSec - startTimeInMicroSec;
}
```

시간 측정 결과 가시화

- 자신의 시계를 구현해 보라.

물리기반 모델링

1.2 운동학

동명대학교
강영민

운동학

- 운동학과 동역학
  - 운동학
    - 힘이 고려되지 않음
  - 위치, 속도, 가속이 시간의 함수로 다뤄짐
    - x(t), v(t), a(t)
- 동역학
  - 힘이 가장 중요한 역할
    - 현재 상태의 힘 f(t)을 계산
    - 가속 계산 a(t) = f(t)/m
    - 속도 갱신 v(t+dt) += a(t)dt
    - 위치 갱식 x(t+dt) += v(t+dt)dt

정역학, 운동학, 동역학, 역학

- 정역학(statics)
  - 평형 상태와 힘의 관계를 연구
- 동역학(kinetics)
  - 운동과 힘의 관계를 연구
- 운동학(kinematics)
  - 관찰된 동작을 이 동작을 유발하는 힘에 대한 고려 없이 연구

classical mechanics

- 역학(dynamics)
  - 정역학 + 동역학

입자와 강체

- 입자
  - 질량을 가진 아주 아주 작은 객체
  - 부피는 무시할 수 있음
    - 예: 로켓 탄도 분석
      - 로켓의 부피는 무시할 수 있음
      - 로켓도 입자로 간주할 수 있음
- 강체 (이상적인 고체)
  - 질량과 부피를 가진 객체
  - 어떤 경우에도 모양이 변하지 않음
  - 강체의 유효한 애니메이션 = 회전과 이동

속도

- 속도(velocity)
  - 벡터
  - 속력(speed): 속도의 크기
  - 속도 = (속력, 방향)
  - 속도의 방향
    - 이동하는 방향
  - 속도의 크기
    - 시간에 대해 이동하는 거리의 비

$$\mathbf{v} = \frac{\Delta \mathbf{s}}{\Delta t}$$

ratio of displacement to time interval

순간 속도

- 시간은 흐른 시간에 대한 이동의 비
  - 시간 간격을 줄이면...
  - 측정하는 순간에 대해 더 정확한 속도를 얻게 됨
  - 시간 간격이 0에 접근할 때 (= 위치를 시간에 대해 미분)
    - 이를 순간 속도라고 함

$$\mathbf{v} = \lim_{\Delta t \to 0} \frac{\Delta \mathbf{s}}{\Delta t} = \frac{d\mathbf{s}}{dt}$$

변위(displacement)

- 속도의 적분

$$\int \mathbf{v} dt = \int d\mathbf{s}$$

  - t1에서 t2 시간 간격 동안의 적분

$$\int_{t_1}^{t_2} \mathbf{v} dt = \int_{s(t_1)}^{s(t_2)} d\mathbf{s}$$

  - 해당 시간 동안 이루어지는 이동량

$$\int_{t_1}^{t_2} \mathbf{v} dt = \mathbf{s}(t_2) - \mathbf{s}(t_1) = \Delta\mathbf{s}$$

$$\mathbf{v} = \lim_{\Delta t \to 0} \frac{\Delta \mathbf{s}}{\Delta t} = \frac{d\mathbf{s}}{dt}$$

$$\mathbf{v} dt = d\mathbf{s}$$

- 만약 속도를 안다면,
  - 미래에 이 입자가 어디에 있을지를 알 수 있음

가속

- 평균 가속
  - 주어진 시간 간격에 대해 변화한 속도의 비

$$\mathbf{a} = \Delta\mathbf{v}/\Delta t$$

- 순간 가속도

$$\mathbf{a} = \lim_{\Delta t \to 0} \Delta\mathbf{v}/\Delta t = d\mathbf{v}/dt$$

- 가속도의 적분

$$\mathbf{a}dt = d\mathbf{v}$$

  - 속도의 변화

$$\int_{t_1}^{t_2} \mathbf{a}dt = \int_{\mathbf{v}(t_1)}^{\mathbf{v}(t_2)} d\mathbf{v} = \Delta\mathbf{v}$$

등가속 운동

- 운동학의 단순한 문제
  - 등가속운동의 예: 중력
  - 중력 가속도
    - 크기: $9.81\ \mathrm{m/s^2}$
    - 방향: 아래쪽 (0,-1,0)
- 가속도가 상수이므로 쉽게 적분할 수 있다.
  - 이 적분을 통해
    - 매 순간 속도의 변화를 알 수 있으며,
    - 매 순간 속도를 계산할 수 있다.

중력 가속 문제

- 중력 가속도: g
  - 속도의 변화와 가속의 적분 사이의 관계

$$\int_{\mathbf{v}(t_1)}^{\mathbf{v}(t_2)} d\mathbf{v} = \int_{t_1}^{t_2} \mathbf{g} dt$$

$$\mathbf{v}_2 - \mathbf{v}_1 = \mathbf{g}(t_2 - t_1)$$

  - t_1에서의 상태를 안다면
  - 쉽게 t_2에서의 상태를 알 수 있다.

$$\mathbf{v}_2 = \mathbf{g}t_2 - \mathbf{g}t_1 + \mathbf{v}_1$$

    - t_1=0이고 t_2=t라면...

$$\mathbf{v}_2 = \mathbf{v}_1 + \mathbf{g}t$$

변위에 대한 함수로의 속도

- 또 다른 미분 방정식

$$\mathbf{v}\frac{d\mathbf{v}}{dt} = \frac{d\mathbf{s}}{dt}\mathbf{a}$$

$$\mathbf{v}d\mathbf{v} = \mathbf{a}d\mathbf{s}$$

- 적분...

$$\int_{\mathbf{v}_1}^{\mathbf{v}_2} \mathbf{v}d\mathbf{v} = \int_{\mathbf{s}_1}^{\mathbf{s}_2} \mathbf{a}d\mathbf{s}$$

$$\frac{1}{2}\mathbf{v}^2\Big|_{\mathbf{v}_1}^{\mathbf{v}_2} = \mathbf{a}\mathbf{s}\Big|_{\mathbf{s}_1}^{\mathbf{s}_2} = \mathbf{g}\mathbf{s}\Big|_{\mathbf{s}_1}^{\mathbf{s}_2}$$

- 속도와 변위 사이의 관계

$$\frac{1}{2}({\mathbf{v}_2}^2 - {\mathbf{v}_1}^2) = \mathbf{g}(\mathbf{s}_2 - \mathbf{s}_1)$$

$${\mathbf{v}_2}^2 = 2\mathbf{g}(\mathbf{s}_2 - \mathbf{s}_1) + {\mathbf{v}_1}^2$$

위치

- 속도와 변위

$$\mathbf{v}dt = d\mathbf{s}$$
$$(\mathbf{v}_1 + \mathbf{g}t)dt = d\mathbf{s}$$

- 적분

$$\int_0^t (\mathbf{v}_1 + \mathbf{g}t)dt = \int_{\mathbf{s}_1}^{\mathbf{s}_2} d\mathbf{s}$$
$$v_1 t + \frac{1}{2}\mathbf{g}t^2 = \mathbf{s}_2 - \mathbf{s}_1$$

- 시간 t에서의 위치

$$\mathbf{s}_2 = v_1 t + \frac{1}{2}\mathbf{g}t^2 + \mathbf{s}_1$$

운동학 시뮬레이션

```
#include "KinematicsSimulator.h"

CKinematicSimulator::CKinematicSimulator() : CSimulator() {}


void CKinematicSimulator::init() {
    initialLoc.set(0,1,0);
    initialVel.set(0,3,0);
    gravity.set(0.0, -9.8, 0.0);
    currentLoc = initialLoc;
    particle.setPosition(currentLoc[0], currentLoc[1], currentLoc[2]);
    particle.setRadius(0.1);
}

void CKinematicSimulator::doBeforeSimulation(double dt, double currentTime) {

}

void CKinematicSimulator::doSimulation(double dt, double currentTime) {
    currentLoc = initialLoc
        + currentTime*initialVel
        + (0.5 * currentTime * currentTime)  * gravity ;
    particle.setPosition(currentLoc[0], currentLoc[1], currentLoc[2]);
    particle.drawWithGL();
}


void CKinematicSimulator::doAfterSimulation(double dt, double currentTime) {

}
```

https://github.com/dknife/201501Lec_GamePhysics/tree/master/GPcode01_03_KinematicParticleExplosion

운동학을 통한 입자 폭발 효과

- KinematicSimulator.cpp

```
#include "KinematicsSimulator.h"

CKinematicSimulator::CKinematicSimulator() : CSimulator() {}
void CKinematicSimulator::init() {

    for(int i=0;i<NUMPARTS;i++)particle[i].randomInit();
}

void CKinematicSimulator::doBeforeSimulation(double dt, double currentTime) { }
void CKinematicSimulator::doSimulation(double dt, double currentTime) {
    for(int i=0;i<NUMPARTS;i++){
        particle[i].simulate(dt, currentTime);
        particle[i].drawWithGL(POINT_DRAW);
    }
}
void CKinematicSimulator::doAfterSimulation(double dt, double currentTime) { }
```

운동학을 이용한 입자 폭발 효과

- Particle.cpp

```
void CParticle::randomInit() {
    double speed = rand()%10000 / 10000.0 + 1.0;
    double theta = 2.0*3.141592 * (rand()%10000 / 10000.0);
    double phi =   2.0*3.141592 * (rand()%10000 / 10000.0);
    double vx,vy,vz;  // spherical coord to cartesian coord
    vy = speed*cos(phi)+2.0;
    vx = speed*cos(theta)*sin(phi);
    vz = speed*sin(theta)*sin(phi);
    initialLoc.set(0,1,0);
    initialVel.set(vx, vy, vz);
    gravity.set(0.0, -9.8, 0.0);
    radius = 0.01;
    currentLoc = initialLoc;
}
void CParticle::simulate(double dt, double et) {
    currentLoc = initialLoc + et*initialVel + (0.5 * et * et)  * gravity ;
    setPosition(currentLoc[0], currentLoc[1], currentLoc[2]);
}
```

결과

물리기반모델링

1.3 역학과 수치적분

동명대학교
강영민

역학

- 역학
  - 힘에 기반하여 운동을 이해
- 중요한 공식 (뉴턴의 제2법칙)
  - $\mathrm{f} = m\mathrm{a}$
- 강체는.... 회전힘 = 회전질량*회전가속

$$\tau = \mathbf{I}\dot{\omega}$$

운동방정식의 적분

- 뉴턴의 운동 제2법칙
  - f = ma
- 다시 말하면...

$$\mathbf{f} = m\frac{d\mathbf{v}}{dt}$$

- dv = ?

$$\frac{\mathbf{f}dt}{m} = d\mathbf{v}$$

$$\int_{t_1}^{t_2} \frac{\mathbf{f}dt}{m} = \int_{\mathbf{v}_1}^{\mathbf{v}_2} d\mathbf{v}$$

힘이 (이 시간 동안) 상수일 경우

$$\frac{\mathbf{f}}{m}(t_2 - t_1) = \mathbf{v}_2 - \mathbf{v}_1$$

$$\frac{\mathbf{f}}{m}\Delta t = \Delta\mathbf{v}$$

초기 조건 문제

- 초기 조건
  - $x(t)$, $v(t)$
  - 시간 t에서의 위치와 속도
- 문제
  - 조금의 시간 dt가 흐른 뒤를 예측
  - 예측의 대상 위치 $x(t+dt)$와 속도 $v(t+dt)$를 구하기
- $x(t+dt)$, $v(t+dt)$를 초기 조건으로 예측을 반복

속도의 갱신

- 속도의 초기 조건
  - v1 = v(t)
- 찾아야 하는 속도
  - v2 = v(t+dt)
- 다음 프레임(t+dt)에서의 속도

$$\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \frac{\mathbf{f}(t)}{m}\Delta t$$

- 혹은

$$\mathbf{v}(t+\Delta t) = \mathbf{v}(t) + \mathbf{a}(t)\Delta t$$

$$\frac{\mathbf{f}}{m}(t_2 - t_1) = \mathbf{v}_2 - \mathbf{v}_1$$

$$\frac{\mathbf{f}}{m}\Delta t = \Delta \mathbf{v}$$

위치의 갱신

- 속도와 위치

$$\mathbf{v} = d\mathbf{s}/dt \qquad\qquad \mathbf{v}dt = d\mathbf{s}$$

- 수치 적분

$$\mathbf{v}(t + \Delta t)\Delta t = \Delta\mathbf{s}$$

- 시간 t에서 가해지는 힘 혹은 가속도를 알 수 있다면
  - 시간 t+dt에서의 위치와 속도를 추정할 수 있다.

실시간 시뮬레이션

- 힘을 계산한다: f
- 가속도를 계산한다: a = f/m
- 정해진 시간 간격이 흐른 뒤의 속도를 계산한다.
  - 오일러 적분

$$\mathbf{v}(t + \Delta t) = \mathbf{v}(t) + \mathbf{a}\Delta t$$

- 시간 간격이 흐른 뒤의 위치도 계산한다.
  - 오일러 적분

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \mathbf{v}(t + \Delta t)\Delta t$$

시뮬레이션 코드 - main.cpp

```
void keyboardFunction(unsigned char key, int x, int y) {
    if (key == 27) exit(0);
    switch (key) {
        case 's':
            if(!myWatch.bRunning()) { Simulator->start(); myWatch.start(); }
            else { myWatch.stop(); Simulator->stop(); }
            break;
        case 'p':
            myWatch.pause(); Simulator->pause();
            break;
        case 'r':
            myWatch.resume(); Simulator->resume();
        default:
            break;
    }
}
void displayFunction(void) {
    …
    // 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);
    glutSwapBuffers();
}
```

시뮬레이션 코드 - Simulator.cpp

```
#include "Simulator.h"
#include <stdlib.h>
#include <stdio.h>
// Constructor
CSimulator::CSimulator(): bRunning(false) {  }
CSimulator::~CSimulator() { }

void CSimulator::actions(double dt, double currentTime) {
    if(bRunning) {
        doBeforeSimulation(dt, currentTime);
        doSimulation(dt, currentTime);
        doAfterSimulation(dt, currentTime);
    }
    visualize();
}
// Control Event Handlers
void CSimulator::start() {
    this->init();
    bRunning = true;
}
void CSimulator::stop() {
    bRunning = false;
    this->clean();
}
void CSimulator::pause() {}
void CSimulator::resume() {}
```

시뮬레이션 코드 - DynamicSimulator.cpp

```
#include "DynamicSimulator.h"

CDynamicSimulator::CDynamicSimulator() : CSimulator() {}

void CDynamicSimulator::init() {
   for(int i=0;i<NUMPARTS;i++)particle[i].randomInit();
}

void CDynamicSimulator::clean() {}

void CDynamicSimulator::doBeforeSimulation(double dt, double currentTime) {}

void CDynamicSimulator::doSimulation(double dt, double currentTime) {
    for(int i=0;i<NUMPARTS;i++)
        particle[i].simulate(dt, currentTime);

}
void CDynamicSimulator::doAfterSimulation(double dt, double currentTime) {}

void CDynamicSimulator::visualize(void) {
    for(int i=0;i<NUMPARTS;i++) {
        particle[i].drawWithGL(POINT_DRAW);
    }
}
```

시뮬레이션 코드 - Particle.cpp

```
void CParticle::simulate(double dt, double et) {
    // Update velocity: Euler Integration of acceleration
    vel = vel + dt*gravity;
    // Update position: Euler Integration of velocity
    loc = loc + dt*vel;

    // collision handling
    if(loc[1]<0) {
        loc.set(loc[0], -0.9*loc[1], loc[2]);
        if(vel[1]<0) vel.set(vel[0], -0.9*vel[1], vel[2]);
    }
    setPosition(loc[0], loc[1], loc[2]);
}
```

결과

물리기반 모델링

1.4 다양한 힘 모델

동명대학교 게임공학과
강영민

힘

- 힘은
  - 운동을 유발한다.
- 역학에서 매우 중요하다.
- 다양한 모델이 존재한다.

다룰 개념들

- 역장(힘의 장): 예 - 중력
- 마찰: 운동에 저항하는 접촉력
- 유체 항력: 유체 내에 움직이는 물체에 가해지는 저항력
- 압력: 단위 면적 당 가해지는 힘
- 부력: 유체에 잠긴 객체를 “위로” 밀어 올리는 힘
- 스프링-댐퍼: 객체를 탄성으로 묶어 놓는 힘
- 회전력: 물체를 회전하게 만드는 “힘의 모멘트”

힘의 장

- 힘의 장(force field)
  - 물체에 가해지는 힘을 표현하는 벡터의 장
- 좋은 예
  - 중력장
  - 전자기장

Gravitational force field

- 만유 인력

$$|\mathbf{f}_u| = Gm_1m_2/r^2$$

  - G: 중력계수 $6.673 \times 10^{-11}(N \cdot m^2)/kg^2$
  - r: 두 질량 사이의 거리
  - m_{1,2}: 각각의 질량
- 지구에서의 중력
  - 지구의 질량 : $5.98 \times 10^{24} kg$
  - 지구 반지름: $6.38 \times 10^6 m$
  - 중력 가속도

$$\frac{Gm_{earth}}{r^2} \simeq (\frac{6.673 \times 5.98}{6.38^2}) \times 10m/s^2 \simeq 9.8034m/s^2$$

마찰력

- 접촉면에 의한 저항력
  - 접촉력
  - 법선 방향으로 가해지는 힘:N이 중요
- 두 종류의 마찰력
  - 정지 마찰력: 최대의 마찰력

$$|\mathbf{f}_{max}| = \mu_s \mathbf{N}$$

  - 운동 마찰력

$$|\mathbf{f}_k| = \mu_k \mathbf{N}$$

마찰계수

- 잘 알려진 표면의 마찰 계수
  - Ms: 정지마찰계수 / Mu: 운동마찰계수

Surface condition	M_s	M_u	% difference
Dry glass on glass	0.94	0.4	54%
Dry iron on iron	1.1	0.15	86%
Dry rubber on pavement	0.55	0.4	27%
Dry steel on steel	0.78	0.42	46%
Dry Teflon on Teflon	0.04	0.04	—
Dry wood on wood	0.38	0.2	47%
Ice on ice	0.1	0.03	70%
Oiled steel on steel	0.10	0.08	20%

유체 항력

- 마찰력과 유사
  - 마찰력은 항력에서 주요한 요소
  - 하지만 마찰력이 전부는 아님
- 천천히 움직이는 객체의 점성 항력: 층류(laminar) 상태
  - $f = -C\,v$
- 빠르게 움직이는 객체의 항력: 난류(turbulence) 상태
  - $f = -C\,v^2$

압력

- 압력은 힘이 아님
  - 압력 = 단위 면적 당 가해지는 힘
  - $F = PA$ (힘 = 압력 x 면적)
  - $P = F/A$
- 압력이 중요한 시뮬레이션 예들
  - 보트, 호버크래프트...

부력

- 유체 내의 서로 다른 압력에 의해 발생
- 수평으로 작용하는 힘의 총합 = 0
- 수직으로 작용하는 힘의 총합 = 아래쪽 면에 작용하는 힘 - 윗면에 작용하는 힘
- F = PA
- 압력: 밀도와 중력의 수
  - 위쪽에 작용하는 압력

$$\mathbf{P}_t = \rho \mathbf{g} h_t$$

  - 아래쪽에 작용하는 압력

$$\mathbf{P}_b = \rho \mathbf{g} h_b$$

부력

- 힘

$$\mathbf{f}_t = \mathbf{P}_t A_t = \rho \mathbf{g} h_t s^2$$
$$\mathbf{f}_b = \mathbf{P}_b A_b = \rho \mathbf{g} h_b s^2$$

- 차이

$$\begin{aligned}\mathbf{f}_b - \mathbf{f}_t &= \rho \mathbf{g} h_b s^2 - \rho \mathbf{g} h_t s^2 \\ &= \rho \mathbf{g}(h_b - h_t) s^2 \\ &= -\rho \mathbf{g} s^3 \\ &= -\rho \mathbf{g} V \quad (V : volume)\end{aligned}$$

스프링 힘

- 후크(Hookd)의 법칙
  - 스프링의 길이를 x만큼 늘이거나 줄이는 데에 필요한 힘은 이 길이에 비례한다.
    - f = -k x
    - k: 스프링 계수
- 아무런 힘이 가해지지 않은 상태에서의 스프링 길이 (휴지 상태 길이) : r
- 현재 스프링의 길이: L
  - 힘의 크기: $$|\mathbf{f}| = k_s(L - r)$$
  - 힘의 방향: 스프링 양쪽에 x1과 x2의 위치에 물체가 달려 있을 때
    - $$\frac{\mathbf{x}_1 - \mathbf{x}_2}{|\mathbf{x}_1 - \mathbf{x}_2|} \qquad -\frac{\mathbf{x}_1 - \mathbf{x}_2}{|\mathbf{x}_1 - \mathbf{x}_2|}$$

댐퍼

- 스프링은 영원히 진동하지 않는다
  - 에너지가 사라짐
  - 간단한 모델
    - 댐핑 힘

$$\mathbf{f}_d = k_d(\mathbf{v}_1 - \mathbf{v}_2)$$

스프링과 댐퍼

$$\mathbf{f}_1 = -(k_s(L - r) + k_d(\mathbf{v}_1 - \mathbf{v}_2) \cdot \frac{\mathbf{L}}{L})\frac{\mathbf{L}}{L}$$

$$\mathbf{f}_2 = -\mathbf{f}_1$$

힘과 토크

- 힘
  - 선 가속도를 일으킨다
- 토크
  - 회전 가속을 일으킨다
- 토크: τ
  - 벡터이다
  - 크기
    - 얼마나 빠르게 회전 속도가 바뀌는지
      - |r x f |
  - 방향
    - 회전 축 = (r x f) / |r x f|

$$\tau = \mathbf{r} \times \mathbf{f}$$

물리기반 모델링

1.5 회전 운동

동명대학교
게임공학과

회전 운동

- 토크: τ
  - 회전 운동에서 힘의 역할

$$\tau = \mathbf{r} \times \mathbf{f}$$

- 회전 운동량
  - 모든 입자의 운동량 모멘트의 합

$$\mathbf{H} = \sum \mathbf{r}_i \times m_i \mathbf{v}_i$$

$$\mathbf{H} = \sum \mathbf{r}_i \times m_i (\omega \times \mathbf{r}_i)$$

$$\mathbf{v}_i = \omega \times \mathbf{r}_i$$

angular velocity

관성 모멘트(회전질량)

- 회전 질량
  - 회전축에 따라 달라짐
  - 축을 중심으로 하는 회전 가속에 저항하는 특성

$$I = \int_m r^2 \mathrm{d}m$$

회전 질량의 예 (3개의 축에 대해)

$$I_{xx} = I_{yy} = (1/4)\, mr^2 + (1/12)\, ml^2 ;\ I_{zz} = (1/2)\, mr^2$$

$$I_{xx} = I_{yy} = (1/4)\, mr^2 + (1/12)\, ml^2 ;\ I_{zz} = (1/2)\, mr^2$$

$$I_{xx} = I_{yy} = (1/4)\, mr^2 + (1/12)\, ml^2 ;\ I_{zz} = (1/2)\, mr^2$$

sphere $I_{xx} = I_{yy} = I_{zz} = (2/5)\, mr^2$

spherical shell $I_{xx} = I_{yy} = I_{zz} = (2/3)\, mr^2$

회전 운동량= 회전질량 x 회전속도

- 선운동량: **G**
  - $\mathbf{G} = m\mathbf{v}$
- 회전운동량

$$\mathbf{H} = \sum \mathbf{r}_i \times m_i(\omega \times \mathbf{r}_i)$$

$$\mathbf{H} = \int \omega \mathbf{r}^2 dm$$

$$= \omega \int \mathbf{r}^2 dm$$

$$= \omega \mathbf{I} = \mathbf{I}\omega$$

회전 질량
(관성 모멘트)

회전 속도(각속도)

회전 운동량의 미분

- $d\mathbf{G}/dt =$ 힘
- $d\mathbf{H}/dt =$ 토크(torque)

$$\frac{d\mathbf{H}}{dt} = \frac{d\mathbf{I}\omega}{dt} = \mathbf{I}\frac{d\omega}{dt} = \mathbf{I}\alpha$$

$$\longrightarrow \quad \sum \tau = \mathbf{I}\alpha \quad \longrightarrow \quad \alpha = \mathbf{I}^{-1}\sum \tau$$

텐서(tensor)

- 텐서
  - 크기와 방향을 가진 수학적 표현
  - 방향에 따라 그 크기가 동일하지 않을 수 있음
  - 다른 방향에 대해 다른 크기를 갖는 물체의 특성을 표현할 때 사용
- 등방성(isotropic) 특성과 이방성(anisotropic) 특성
  - 등방성: 모든 방향으로 동일한 특성
  - 이방성:방향에 따라 달라지는 특성
- 관성 모멘트
  - 관성 텐서(3차원)
  - 아홉 개의 요소가 모든 방향으로의 특성을 표현할 수 있음
  - 회전 축에 따라 달라지는 강체의 특성을 표현

물리기반 모델링

1.6 강체 - 2차원

동명대학교
강영민

강체의 운동

- 입자와 강체의 차이
  - 입자: 회전이 없음
  - 강체: 회전
- 강체의 운동
  - 질량 중심의 선운동 (입자와 동일)
  - 회전 운동
    - 토크 τ(선운동에서 힘 f와 같은 작용)
      - 각 속도 ω (속도 **v**와 같은 역할)
      - 각 가속도 $\dot{\omega}$ (가속도 **a**와 같은 역할)

지역 좌표계

- 회전
  - 지역 좌표계의 원점을 중심으로 회전

- 2차원 강체의 회전
  - z 축 회전
- 회전은 원래의 상태에서 회전된 각을 표현하는 하나의 실수 Ω 로 표현 가능

각 속도와 각 가속도

- 선속도 = 시간에 대한 위치의 변화 비
- 각 속도 = 시간에 대한 회전 각의 변화 비
  - >> $$\omega = \frac{d\Omega}{dt}$$
- 각 가속도

$$\dot{\omega} = \frac{d\omega}{dt}$$

회전에 의한 선속도

- 각도 Ω만큼의 회전
  - 지역 좌표 중심에서 r만큼 떨어진 위치는 원호 c를 따라 이동
- 간단한 관찰 $$c = r\Omega$$
- 미분하면.... $$dc/dt = rd\Omega/dt = r\omega$$

$$v = r\omega$$

- 가속
  - a = dv / dt

$$a = r\dot{\omega}$$

2차원 강체 시뮬레이션

- 상태 $(\mathbf{x}, \mathbf{v}, \Omega, \omega)$
- 관성
  - 질량 m : 선 운동에 대한 저항
  - 관정 모멘트 I : 회전 운동에 대한 저항
- 시뮬레이션
  - 힘과 토크를 계산 $\mathbf{f}, \tau$

$$\tau = \mathbf{r} \times \mathbf{f}$$

2차원에서는....

$$\mathbf{f} = (f_x, f_y, 0)$$
$$\mathbf{r} = (r_x, r_y, 0)$$
$$\tau = (0, 0, \tau_z)$$

적분

- 선운동

$$\mathbf{v}(t+dt) = \mathbf{v}(t) + \frac{\mathbf{f}}{m}dt$$
$$\mathbf{x}(t+dt) = \mathbf{x}(t) + \mathbf{v}(t+dt)dt$$

- 회전운동

$$\omega(t+dt) = \omega(t) + I^{-1}\tau dt$$
$$\mathbf{\Omega}(t+dt) = \mathbf{\Omega}(t) + \omega(t+dt)dt$$

- 2D
  - I: 스칼라 … $I^{-1} = \frac{1}{I}$

구현

- Dynamic Simulator

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

    hover.simulate(dt);
}

void CDynamicSimulator::visualize(void) {

    hover.draw();

}

void CDynamicSimulator::control(unsigned char key) {
    int engineNumber = (int) (key-'1');
    hover.switchEngine(engineNumber, hover.isEngineOn(engineNumber)?false:true);
}

CVec3d CDynamicSimulator::getCameraPosition(void) {
    CVec3d loc;
    loc = hover.getLocation();
    return loc;
}
```

Hovercraft.h

```
enum ENGINE_NUMBER {
    LEFT_THRUST,
    RIGHT_THRUST,
    RIGHT_SIDE,
    FRONT_BRAKE,
    LEFT_SIDE,
    NUMBER_OF_ENGINES
};

class CHovercraft {
    double mass;
    double inertia;
    CVec3d loc;
    CVec3d vel;
    CVec3d force;
    double angle;
    double aVel;
    double torque;

    CVec3d r[NUMBER_OF_ENGINES];
    CVec3d fLocal[NUMBER_OF_ENGINES];
    bool   on[NUMBER_OF_ENGINES];
    CVec3d localVectorToWorldVector(const CVec3d &lV);
public:
  …
    void draw(void);
    void switchEngine(int engineNumber, bool switch_state);
    bool isEngineOn(int engineNumber);
    void simulate(double dt);
    void setLocation(CVec3d location);
    CVec3d getLocation(void);
};
```

Hovercraft.cpp - 생성자

```
CHovercraft::CHovercraft() :
mass(1.0), inertia(1.0), angle(0.0), aVel(0.0), torque(0.0) {
    loc.set(0.0, 0.0, 0.0);
    vel.set(0.0, 0.0, 0.0);
    force.set(0.0, 0.0, 0.0);

    r[LEFT_THRUST].set(-1.0, -1.0, 0.0);
    r[RIGHT_THRUST].set(1.0, -1.0, 0.0);
    r[LEFT_SIDE].set(-1.0, 1.0, 0.0);
    r[RIGHT_SIDE].set(1.0, 1.0, 0.0);
    r[FRONT_BRAKE].set(0.0, 1.5, 0.0);

    fLocal[LEFT_THRUST].set( 0.0, 1.0, 0.0);
    fLocal[RIGHT_THRUST].set(0.0, 1.0, 0.0);
    fLocal[LEFT_SIDE].set(   1.0, 0.0, 0.0);
    fLocal[RIGHT_SIDE].set( -1.0, 0.0, 0.0);
    fLocal[FRONT_BRAKE].set( 0.0,-1.0, 0.0);

    for (int i=0; i<NUMBER_OF_ENGINES; i++) on[i] = false;
}

CHovercraft::~CHovercraft() {

}
```

Hovercraft.cpp - 시뮬레이션 부분

```
void CHovercraft::simulate(double dt) {
    force.set(0.0, 0.0, 0.0);
    torque = 0.0;

    // rigid body
    CVec3d fWorld;
    CVec3d torqueVec;
    for (int i=0; i<NUMBER_OF_ENGINES; i++) {
        if(on[i]) {
            fWorld = localVectorToWorldVector(fLocal[i]);
            force = force + fWorld;
            torqueVec = r[i]*fLocal[i];
            torque += torqueVec[2];
        }
    }
    // drag force
    double kd = 0.5;
    force  = force -kd*vel;
    torque += -kd*aVel;

    // numerical integration
    vel = vel + (dt/mass)*force;
    loc = loc + dt * vel;
    aVel = aVel + (dt/inertia)*torque;
    angle = angle + dt * aVel;
}
```

애니메이션 결과

- https://www.youtube.com/watch?v=xbu_-VP7Ed0
- http://goo.gl/s8TTAi

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

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

```
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 에너지 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

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

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

```
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();
```

```
   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

```
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

```
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

```
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

```
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