게임프로그래밍

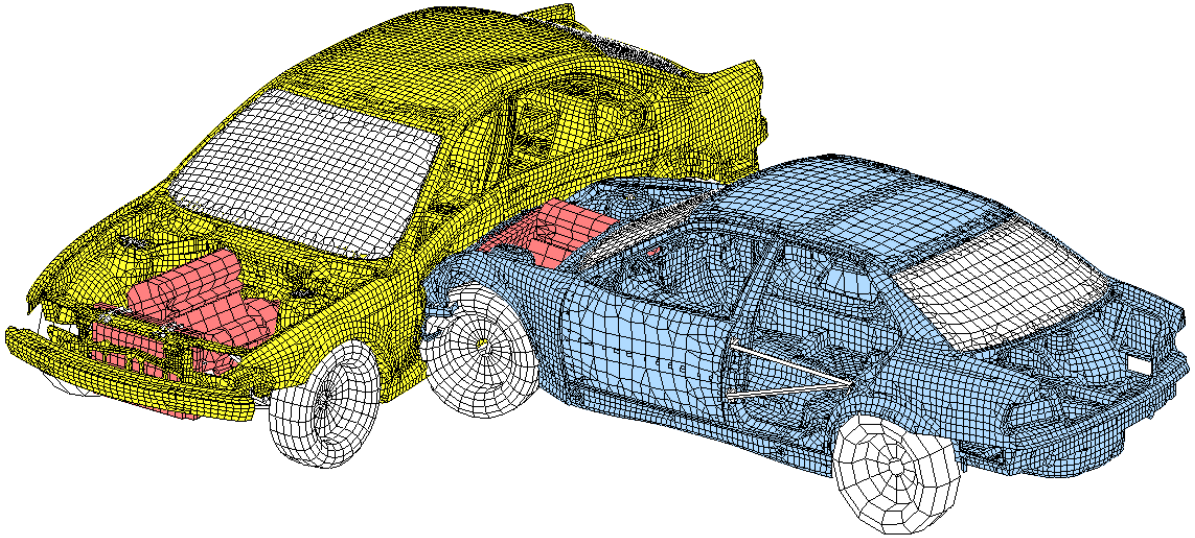
게임 물리

박종승

Dept. of CSE, Incheon Nat. Univ.
jong@inu.ac.kr
http://ecl.inu.ac.kr

목차

- 물리 개요
- 운동의 기초
- 역학
- 적분법



게임 물리 개요

- 게임 세계에서의 물리
  - 게임의 현실감이 크게 향상됨 → 플레이어가 더 몰입할 수 있음
  - 물리 현상을 실세계와 동일하게 구현할 필요는 전혀 없음
    - 비슷하게만 하면 됨
- 물리 엔진
  - 상용 물리엔진 : PhysX, Havok

용어

- 역학(mechanics; 力學)이란
  - 힘을 받고 있는 물체의 반응(운동) 상태를 다루는 분야
  - 역학의 분야
    - 강체역학(mechanics of rigid body)*
    - 변형체 역학(mechanics of deformable bodies)
    - 유체역학(mechanics of fluids)
- 강체 역학
  - 물체의 변형을 고려하지 않음
  - 정역학(statics)
    - 작용하는 힘이 균형을 이루어 물체가 정지 또는 등속운동(가속도 0)
  - 동역학(dynamics)*
    - 강체가 가속 혹은 감속 운동
- 동역학
  - 물체가 왜 움직이는가를 설명
  - 운동학(kinematics) 와 운동역학(kinetics) 로 구성됨

용어'

- 운동학(kinematics)
  - 내재된 힘을 고려하지 않고 운동(motion)을 다루는 분야
    - 물체가 시공간에서 어떻게 움직이는지를 설명함.
      - the study of geometry of motion
      - 즉 변수들(위치, 속도, 가속도, 자세, 각속도, 각가속도 등)의 시간에 따른 변화와 기하학적인 관계만을 다룸.
    - 게임의 캐릭터 애니메이션
  - 전진과 역(forward vs inverse)
    - 전진: joint들의 behavior에 따른 자세(좌표값)를 결정
    - 역: 특정 pose를 얻기 위한 joint들의 behavior(회전각도)를 결정
- 운동역학(kinetics)
  - 힘(force)을 받은 물체의 운동(motion)을 설명하는 학문
    - 물체의 가속운동을 힘의 개념으로 이해
    - 예: 운동방정식, 일과 에너지의 관계, 충격량과 운동량의 관계
  - 전진과 역(forward vs inverse)
    - 전진: 적용된 force로 인해 발생하는 motion을 계산
    - 역: 원하는 motion을 생성하기 위해 필요한 force를 계산

위치 함수

- 위치 함수(position function): **x**(t)
  - 물체의 위치를 time의 함수로 표현
  - 가정: 초기 위치는 알려져 있음, time은 초기 위치에 상대적으로 측정됨
- 물체의 위치를 속도(velocity)로 표현
  - 가정: 일정 속도 $\mathbf{v}_0$로 직선운동, 시간 t=0에서의 물체의 위치 : $\mathbf{x}_0$
  - 임의의 시간 t에서의 위치 : $\mathbf{x}(t)=\mathbf{x}_0+\mathbf{v}_0t$
- 속도 함수: **v**(t)
  - 물체의 3D 속도를 time의 함수로 표현
  - 위치 함수를 time에 대해 미분한 것임

$$\mathbf{v}(t)=\dot{\mathbf{x}}(t)=\frac{d}{dt}\mathbf{x}(t)$$

  - 속도가 일정한 경우 : 하나의 상수로 둠 : $\mathbf{v}(t)=\mathbf{v}_0$
  - 속도가 변하는 경우(일정한 가속) : $\mathbf{v}(t)=\mathbf{v}_0+\mathbf{a}_0t$, $\mathbf{a}_0$를 가속도(acceleration)라고 함

위치 함수'

- 가속도 함수: **a**(t)
  - 물체의 3D acceleration을 time의 함수로 표현
  - 속도 함수를 time에 대해 미분한 것임

$$\mathbf{a}(t) = \dot{\mathbf{v}}(t) = \ddot{\mathbf{x}}(t) = \frac{d^2}{dt^2}\mathbf{x}(t)$$

- 위치 구하기
  - time t_1에서 t_2 사이에 물체가 이동한 거리 d : velocity를 적분

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

  - 일정한 가속도로 움직이는 물체의 위치 : **x**(t)

$$d = \int_{t_1}^{t_2} (\mathbf{v}_0 + \mathbf{a}_0 t)dt = \mathbf{v}_0 t + \frac{1}{2}\mathbf{a}_0 t^2$$

$$\longrightarrow \quad \mathbf{x}(t) = \mathbf{x}_0 + \mathbf{v}_0 t + \frac{1}{2}\mathbf{a}_0 t^2$$

포물선 운동

- 중력(force of gravity)에 의해서만 영향을 받는 물체의 운동
  - 위쪽이 world space의 +z라고 가정

    → 중력 가속도 **g**=[0,0,-g]
    - 여기서 g는 중력 상수로, 지표면에서 약 9.8 m/s^{2}임
    - 중력장 내의 물체는 아래쪽(지구 중심쪽)으로 mg의 힘을 받음. m은 질량(mass)임
  - 시간 t=0에서 초기 위치가 $\mathbf{x}_0$이고, 초기 속도가 $\mathbf{v}_0$인 발사체(projectile)의 위치

$$\mathbf{x}(t) = \mathbf{x}_0 + \mathbf{v}_0 t + \frac{1}{2}\mathbf{g}t^2 \longrightarrow \begin{aligned} x(t) &= x_0 + v_x t \\ y(t) &= y_0 + v_y t \\ z(t) &= z_0 + v_z t - \frac{1}{2}gt^2 \end{aligned} \qquad \mathbf{x}_0 = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix}, \mathbf{v}_0 = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

  - 발사체가 최대 높이에 도달하는 시간 t

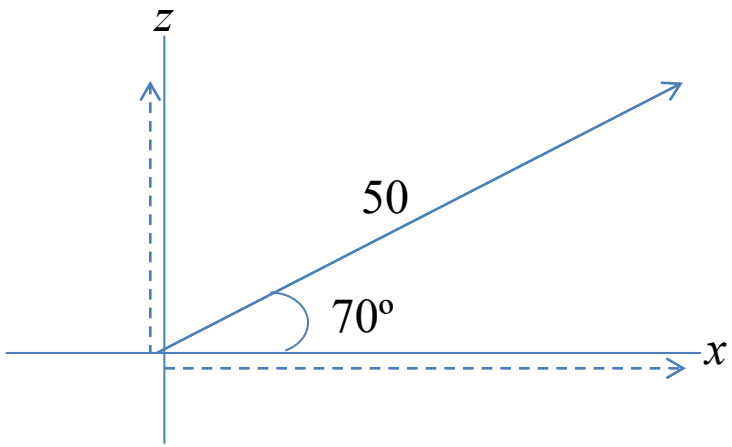
$$\dot{z}(t) = v_z t - gt = 0 \longrightarrow t = \frac{v_z}{g}$$

  - 발사체가 도달할 수 있는 최대 높이 h $\longrightarrow$ $h = z_0 + \frac{v_z^2}{2g}$

포물선 운동'

- 속도(velocity)와 속력(speed)의 구분
  - 속도는 vector, 속력은 scalar

- 예: speed 50m/s로 70°로 발사체를 발사. 속도 $v=[v_x,v_y,v_z]$
  - 상하로의 속력 $v_z=50*\sin 70°$
  - 위에서 내려다 본 모습(xy평면)에서의 거리가 관심이 있을 때,
  - x축으로 발사했다고 가정: $v_x=50*\cos 70°$, $v_y=0$



포물선 운동''

- 발사체의 이동 수평 거리
  - 발사체가 원래의 높이로 내려올 때까지 날아간 수평 거리

$$z(t) = z_0 + v_z t - \frac{1}{2} g t^2 = z_0 \longrightarrow t = 0 \quad \text{또는} \quad t = \frac{2v_z}{g}$$

$$x(t) = x_0 + v_x t \quad \text{에 t를 대입} \longrightarrow \boxed{r = \frac{2v_x v_z}{g}}$$

- 발사 각도
  - 발사체를 최대로 높이 올릴 수 있는 발사각도:
    - 단, 발사될 때의 초기 속력 s가 주어졌을 때,

$$h = z_0 + \frac{v_z^2}{2g} \longrightarrow h = z_0 + \frac{(s\sin\alpha)^2}{2g} \longrightarrow \boxed{\alpha = \sin^{-1}\left(\frac{1}{s}\sqrt{2g(h - z_0)}\right)}$$

  - 원하는 도달거리 r를 가기 위한 발사각도:

$$r = \frac{2v_x v_z}{g} \longrightarrow r = \frac{2(s\cos\alpha)(s\sin\alpha)}{g} = \frac{s^2}{g}\sin 2\alpha \longrightarrow \boxed{\alpha = \frac{1}{2}\sin^{-1}\frac{rg}{s^2}}$$

    - 주의: α가 해로 구해졌으면 π/2-α도 해가 됨 (sin(π-α)=sinα이므로)

힘의 기초

- 힘(force)
  - 힘이 가해지면 → 운동량(momentum)과 가속도(acceleration)가 변함
  - 여러 force들을 하나의 전체 force로 더할 수 있음
    - $f_{total}=\sum f_i$
- 뉴턴의 법칙(Newton's Laws)
  - 제 1 법칙 (관성의 법칙)
    - A body at rest tends to stay at rest, and a body in motion tends to stay in motion, unless acted upon by some force.
  - 제 2 법칙 (가속도의 법칙)
    - Forces lead to changes in momentum and therefore accelerations.
      - $f=ma$
  - 제 3 법칙 (반작용의 법칙)
    - Every action has an equal and opposite reaction.
      - $f_{ij}=-f_{ji}$

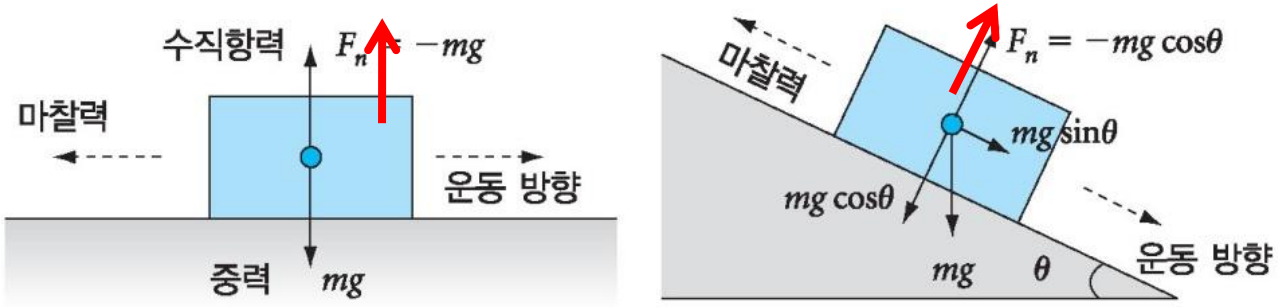
중력

- 중력(gravity)
  - Gravity near Earth's surface is constant
    - $f=mg$ $(g = -9.8\ m/s^2)$
      - f의 단위: $N=kg \cdot m/s^2$
  - Gravity for distant objects
    - $f=GmM/r^2$ $(G=6.673\times10^{-11})$
      - m,M:질점의 질량(kg), r:거리(m)
      - G의 단위: $Nm^2/kg^2=m^3/(kg \cdot s^2)$


The apple tree at Newton's birthplace

수직항력

- 수직항력(normal force) F_n
  - 표면이 물체에 가하는 반작용 힘(reaction force)
    - 물체가 표면에 가하는 힘 (일반적으로 중력)의 법선 성분의 반대
  - 항상 표면에 수직인 방향으로 작용
  - 경사가 없으면, F_n = -mg
  - 경사가 있으면, F_n = -mg·cosθ



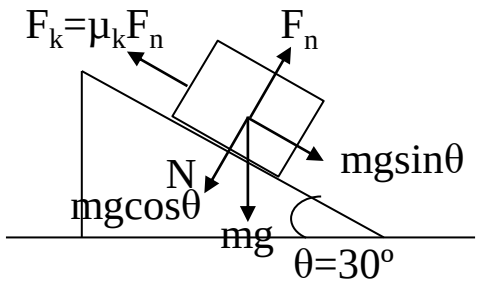
[그림 17-1] 수직항력

마찰력

- 마찰력(force of friction)
  - 마찰력: 표면 상의 물체의 운동을 방해하는 힘
    - 물체에 가해지는 접선 방향 힘의 반대 방향으로 작용
  - 마찰력의 크기: 표면 특성에 의존됨.
    - 물체의 질량에 비례. 접촉 면적과는 무관함.
  - 마찰력의 방향: 운동 방향과 반대방향
  - 마찰력의 계산: 수직 항력으로부터 계산됨
    - $F_f = \mu F_n$ (μ=마찰계수)
    - 마찰계수는 표면의 특성에 따라서 정해짐. >0, 일반적으로 <1.
- 마찰력의 종류
  - 운동마찰력(kinetic friction) : $F_k = \mu_k F_n$
    - 접촉한 두 표면이 서로 상대적으로 움직이는 경우
  - 정지마찰력(static friction) : $F_s = \mu_s F_n$
    - 움직이지 않는 경우
  - 하나만 적용됨

운동마찰

- 운동마찰
  - 서로 상대적으로 움직이는 두 표면 사이에서 발생하는 힘
  - 각자의 운동에 대한 저항으로 작용함
  - 운동마찰력: $F_k = \mu_k F_n$
    - 운동마찰계수 μ_k: >0. 표면의 재질에 따라 다름.



- 예
  - 경사면을 미끄러져 내려가는 물체의 가속도? (단 μ_k=0.5, m=10kg)
    - F(물체에 가해지는 총 힘) = F_g(평행면으로의 중력) + F_k(운동마찰)
  - 중력=mg을 분해
    - 경사면과 평행인 성분은 → F_g=mgsinθ
    - 경사면에 수직인 중력 성분은 → N=mgcosθ → F_n = -mgcosθ
  - 운동마찰력 $F_k=\mu_k F_n=-\mu_k mg\cos\theta$
  - a(가속도)=F/m=$(F_g+F_k)/m=g\sin\theta-\mu_k g\cos\theta$
    - 즉, a=$(9.8m/s^2)*(1/2)-0.5*(9.8m/s^2)*(\sqrt{3}/2)$=(약)$0.656m/s^2$

정지마찰

- 정지마찰
  - 표면이 정지된 물체를 움직이지 못하도록 붙들고 있는 힘
  - 정지마찰력: $F_s = \mu_s F_n$
    - 정지마찰계수 μ_S: >0. 표면의 재질에 따라 다름.
  - F_s와 F_k의 관계
    - 물체에 가해지는 힘이 F_s의 최대값을 넘는 순간, 물체가 움직이기 시작함
    - 그때부터는 F_s가 사라지고, F_k가 작용함
    - 참고: 대체로 $F_k < F_s$임
      - 정지된 물체를 움직이게 하는 것이 움직이는 물체를 계속 움직이게 하는 것보다 더 힘들다는 의미
- 예
  - 수평면에 물체가 놓여 있음. 평면을 조금씩 기울일 때 물체가 미끄러지기 시작하는 각도? (단 $\mu_s=0.5$)
    - 경사면의 내리막과 평행인 중력 성분 F_g + 정지마찰력 F_s = 0 시점임
      → $mg\sin\theta + (-\mu_s mg\cos\theta) = 0 \rightarrow \theta = \tan^{-1}\mu_s$
    - 즉, $\theta = 26.6^\circ$

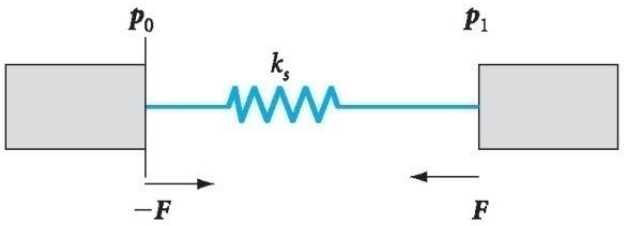
마찰계수

- 경험치

위 물체의 재질	아래 표면의 재질	마찰계수			
		마른 상태		미끈거리는 상태	
		정지	운동	정지	운동
알루미늄	알루미늄	1.1	1.4	0.3	
알루미늄	강철	0.61	0.47		
차 브레이크 패드	주철 브레이크 디스크	0.4		0.2	
놋쇠	강철	0.35	0.3	0.19	
벽돌	나무	0.6			
주철	주철	1.1	0.15	0.2	0.07
주철	나무		0.49		0.075
구리	주철	1.05	0.29		
구리	구리	1.0		0.08	
구리	강철	0.53	0.36		0.18
구리	유리	0.68	0.53		
다이아몬드	다이아몬드	0.1		0.1	
유리	유리	0.94	0.4	0.1	0.09
유리	금속	0.5		0.2	
유리	니켈	0.78	0.56		
사람의 관절	사람의 관절	0.01	0.003		
얼음	얼음	0.1	0.03		
가죽	금속	0.6		0.2	
가죽	나무(결을 따라)	0.61	0.52		
나일론	나일론	0.2			
나무	나무(결을 따라)	0.62	0.48		
나무	나무(결을 가로질러)	0.54	0.32		0.072
고무	아스팔트	0.85	0.67	0.53	
고무	콘크리트	0.9	0.68	0.7	0.58
고무	얼음	0.15			
은	은	1.4		0.55	
강철	주철	0.4	0.23	0.183	0.133
강철	얼음	0.1	0.05		
강철	납	0.95	0.95	0.5	0.3
강철	강철	0.74	0.57	0.15	0.1
스키	눈	0.14	0.05	0.14	0.1
나무	나무	0.42	0.30	0.2	
나무	콘크리트	0.62			

단순한 스프링 운동

- 스프링 운동
  - 비강체(non-rigid) 물체의 모델링
    - rope, string, cloth 등 뿐만 아니라 젤리 방울 등의 모델링 등 다양하게 쓰임
  - 충돌 반응: 물체들이 서로 실감나게 충돌하도록 구현
  - 관절의 시뮬레이션: rigid body의 움직임을 제약하는 joint의 생성
- 스프링의 힘 (Hooke's Law)
  - $F = - k_s x$
    - $x=p_1-p_0$, p_0,p_1: 스프링에 연결된 각 물체의 위치벡터
      - 두 끝점이 닿을 때까지 당기는 경우임
    - k_s=강성계수(stiffness coefficient)
  - $F = - k_s(||x||-d)(x/||x||)$
    - 두 지점의 거리가 d보다 가까우면 밀어내고, d보다 멀면 당기도록 함

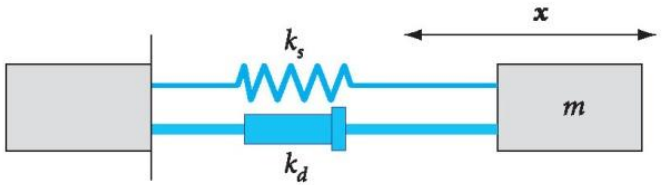


[그림 17-2] 스프링 시스템

감쇠된 스프링 운동

- 감쇠력(damping force)
  - energy loss를 도입하여, 스프링의 무한히 oscillate하지 않도록 함
  - $F = -k_d v$
    - $v=v_1-v_0$, v_0,v_1: 각 위치에서도 속도벡터
    - k_d=점성 감쇠계수(viscous damping coefficient)
- 감쇠된 스프링 시스템(damped spring mass system)
  - 스프링 힘 + 감쇠력
  - $F = -k_s x - k_d v$
  - X 대신 $(\|x\|-d)(x/\|x\|)$ 를 사용할 수 있음
- 위치 구하기
  - F=ma 이므로, $-k_s x - k_d v = ma$

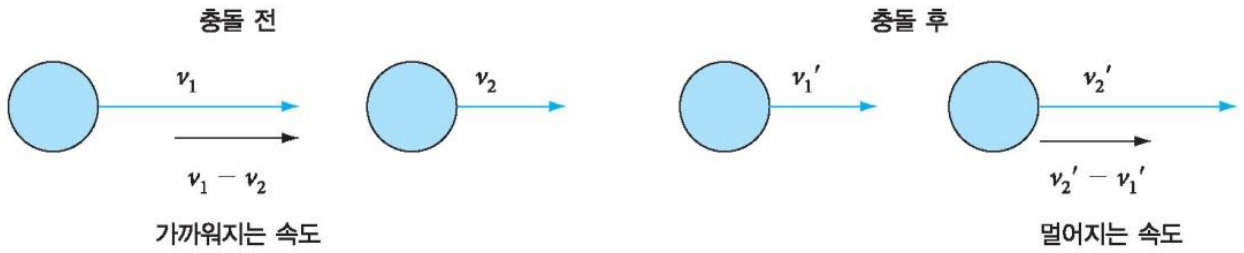
    → $mx'' + k_d x' + k_s x = 0$ (미분방정식)



[그림 17-3] 스프링 댐퍼 시스템

충돌에서의 운동량 보존

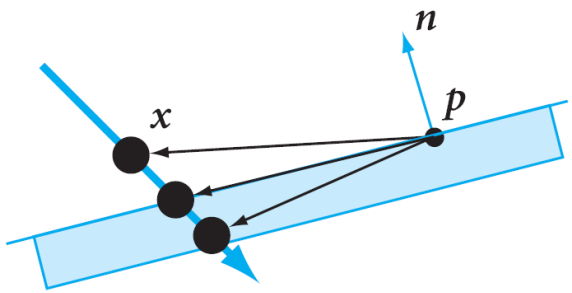
- 운동량 p = mv (kg·m/s)
- 운동량 보존 법칙
  - $m_1v_1+m_2v_2=m_1v_1'+m_2v_2'$



[그림 17-4] 운동량 보존

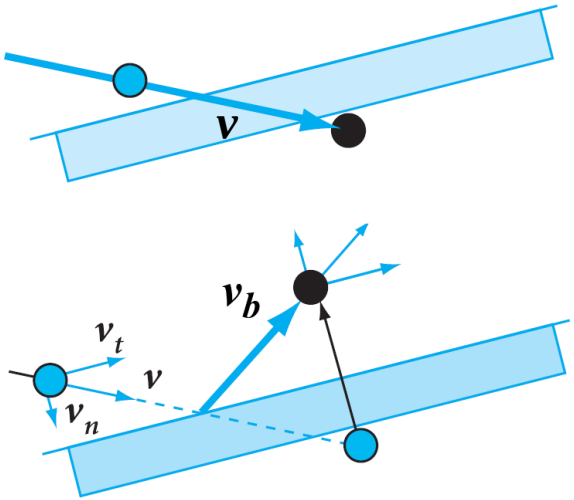
입자와 평면의 충돌 검사

- Particles often bounce off surfaces.
  - Need to detect when a collision has occurred.
  - Need to determine the correct response to the collision.
- 충돌 검사
  - General Collision problem is complex:
    - Particle/Plane Collision (그나마 쉬운 편)
    - Plane/Plane Collision
    - Edge/Plane Collision
- 입자와 평면의 충돌 검사
  - **p**=any point on the plane
  - n=normal pointing on the "legal" side of the plane.
  - x=position of point we want to examine.
  - For **(x–p)·n**
    - If > 0 then x is on legal side of plane.
    - If = 0 then x is on the plane.
    - If **<** **0** then x is on the wrong side of plane



입자와 평면의 충돌 반응

- Collision Response
  - particle이 평면을 관통하는 경우(실제로는 불가능)를 다룸
  - 만약 particle x가 plane의 틀린 쪽에 있다면, 면의 표면으로 옮긴 후, collision response를 계산함
- 방법
  - v=current velocity
  - n=normal to the collision plane
    - v의 normal 성분: $v_n=(v\cdot n)n$
    - v의 tangential 성분: $v_t=v-v_n$
  - v_b=bounced response
    - $v_b=v_t-v_n$
    - $v_b=(1-k_f)*v_t-(k_r*v_n)$
    - k_r=복원계수(coefficient of restitution)
      - 표면이 얼마나 잘 튀는 지; 1=perfectly elastic; 0=stick to wall.
    - k_f=마찰계수(coefficient of friction)
      - 접선 성분이 얼마나 약해지는 지; 1=stops in its tracks. 0=no friction.



물리상태의 계산

- 물리상태의 계산
  - 물체의 이전 물리 상태(위치, 속도 등)와 가해진 힘을 알면,
  - 시간에 대한 "적분"을 통해, 이후의 물체의 상태를 결정할 수 있음
- Motion Equation

$F = ma \rightarrow a = F/m \rightarrow$ force를 알면 acceleration를 구할 수 있음

$dv/dt = a \rightarrow dv = a\,dt \rightarrow$ a를 시간에 대해 적분하여 velocity를 구할 수 있음

$dx/dt = v \rightarrow dx = v\,dt \rightarrow$ v를 시간에 대해 적분하여 position를 구할 수 있음

  - 따라서, 힘을 알면, 모두 구할 수 있음
    - acceleration = force divided by mass
    - change in velocity over time = acceleration * dt
    - change in position over time = velocity * dt
  - 구현은 어떻게? → 적분

```
update_rigidbody(float dt) {
  compute_forces();
  compute_acceleration();
  compute_velocity(); //integration
  compute_position(); //integration
}
```

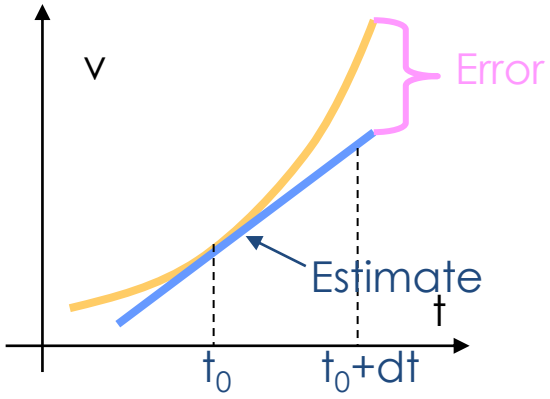
오일러 적분법

- 적분 – 게임 물리에서 중요한 연산임
- 오일러 적분법
  - 각 timestep마다 변화들을 이전 값에 더해 나가면 됨
  - 가장 기본적인 적분 방법임
  - $v=v_0+a\Delta t$, $x=x_0+v\Delta t$
  - 변화율이 상수일 때는 100% 정확함
  - 변화율이 시간에 따라 변할 때는 에러가 존재함
    - 예: 위의 예에서 가속도가 상수이므로 속도는 항상 정확함. 그러나, 속도는 상수가 아니므로 속도가 timestep 사이에서 변화하므로 위치는 부정확함

```
float t = 0; //현재 시간
float dt = 1; //시간 간격(timestep)
float velocity = 0; //초기 속도
float position = 0; //초기 위치
float force = 10;
float mass = 1;
while (t<=10) {
  position += velocity * dt; //현재의 velocity를 사용하여 위치 갱신
  velocity += (force/mass) * dt;
  t = t + dt;
}
```

오일러 적분법'

- 오일러 적분법의 에러
  - 구현은 단순하지만, 에러가 심각한 수준임
  - 예: a=10인 경우 10초 후의 이동거리:
    - 초기속도=0 가정.
    - 실제 이동거리=500m
    - 계산된 이동거리=470m



- 한계
  - timestep이 커질 수록 에러가 커짐
  - 해결책: timestep을 작게 함 → 에러가 상당히 줄어들지만, 여전히 있음
- 개선
  - 제안: 이 방법을 사용하지 말것!
  - 더 좋은 방법:
    - Adaptive Euler Method
    - Midpoint Method
    - Implicit Euler Method
    - Runge Kutta Method
  - 권장: 4차 Runge-Kutta 적분법의 구현 – 권장

Runge Kutta 적분법

- 권장하는 적분법: RK4
  - 4차 Runge Kutta 적분 방법
  - RK4는 미분방정식을 5차 Taylor's series expansion한 것에 대한 에러에 해당함
    → 매우 정확하고 미분에도 안정적임
- 구조체:

```
struct State{ // 물체의 상태를 위한 구조체
  float x; // position
  float v; // velocity
};
struct Derivative{ // 상태 값들의 derivative들을 저장하기 위한 구조체
  float dx; // derivative of position: velocity
  float dv; // derivative of velocity: acceleration
};
```

- 새로운 derivate의 재계산

```
Derivative evaluate(const State &initial, float t, float dt, const Derivative &d) {
  // derivative를 이용하여 physics state를 갱신하고(Euler Integration 방법)
  State state;
  state.x = initial.x + d.dx*dt;
  state.v = initial.v + d.dv*dt;
  // 갱신된 physics state를 사용하여 derivative를 재계산
  Derivative output; //새로운 derivative를 계산
  output.dx = state.v;
  output.dv = acceleration(state, t+dt);
  return output;
}
```

Runge Kutta 적분법'

- 응용에 따른 가속도 계산

```
float acceleration(const State &state, float t) {
//Calculate a spring and damper force and the acceleration
//assuming unit mass.
//The code is completely simulation dependent.
  const float k = 10;
  const float b = 1;
  return -k * state.x - b*state.v;
}
```

- RK4 적분

```
void integrate(State &state, float t, float dt) {
//4개의 derivative 샘플들 a,b,c,d을 계산
  Derivative a = evaluate(state, t);//t에서의 derivative
//:note: overloaded 'evaluate' just returns derivative at current time 't'
  Derivative b = evaluate(state, t, dt*0.5f, a);//a를 이용
  Derivative c = evaluate(state, t, dt*0.5f, b);//b를 이용
  Derivative d = evaluate(state, t, dt, c);//c를 이용
//가장 적합한 derivative를 계산: Taylor Series expansion로부터 유도된 derivative들의 가중치 합
  const float dxdt = 1.0f/6.0f * (a.dx + 2.0f*(b.dx+c.dx) + d.dx);
  const float dvdt = 1.0f/6.0f * (a.dv + 2.0f*(b.dv+c.dv) + d.dv)
  state.x = state.x + dxdt * dt;
  state.v = state.v + dvdt * dt;
}
```

운동량 기반 접근

- 이전의 가속도 기반 방법
  - 1. divide force by mass → acceleration
  - 2. integrate acceleration → the velocity
  - 3. integrate velocity → position
- 운동량(momentum): $\mathbf{p} = m\mathbf{v}$ (velocity*mass)
  - 운동량 기반 방법이 편리한 이유: 회전을 다루기도 적합
  - $d\mathbf{p}/dt = \mathbf{F}$ (the derivative of momentum is force)
  - $\mathbf{v} = \mathbf{p}/m$ (velocity equals momentum divided by mass)
  - $d\mathbf{x}/dt = \mathbf{v}$ (the derivative of position is velocity)
- 운동량 기반 방법
  - 1. integrate force → momentum
  - 2. momentum/mass → velocity
  - 3. integrate velocity → position

```
update_rigidbody(float dt) {
  force = compute_total_force();
  momentum += force*dt;
  velocity = momentum / mass;
  position += velocity*dt;
}
```

시간간격 조절

- 물리 시뮬레이션을 화면에 표시하기
  - 초기 상태(위치, 속도 등)에서 시작
  - 시간에 따라 motion수식을 integrate하여 상태를 갱신
- 시간간격(timestep)의 결정?
- 단순한 구현: 시간간격을 고정
  - 시스템의 속도를 고려하지 않음

```
float t = 0.0f;
float dt = 0.01f; //step ahead 1/100th of a second every step
while (!quit)
{
  integrate(state, t, dt);
  render(state);
  t += dt;
}
```

시간간격 조절'

- 개선한 방식: 타이머방식
  - 시스템의 속도와 무관

```
float t = 0.0f;
float currentTime = time();
while (!quit) {
  float newTime = time();
  float dt = newTime - currentTime;
  currentTime = newTime;
  integrate(state, t, dt);
  t += dt;
  render(state);
}
```

- 문제점과 해결책
  - 문제점: 시간간격이 클 수록 integration이 부정확해짐
  - simulation timestep과 display framerate와의 tradeoff
    - 한쪽 시간간격을 좁히면 다른쪽의 시간간격이 늘어남
  - 예: display framerate은 50/100fps, physics simulation은 100/50fps?

→ 해결책: physics simulation과 display framerate을 **decouple**해야 함

물리 상태 갱신의 분리

- Decoupling Physics timestep
  - 구현: time accumulator를 사용 → framerate와 독립적으로 simulation
    - Undersampling(physics framerate이 더 큰 경우), Oversampling(display framerate이 더 큰 경우)을 모두 해결함
    - 추가적인 고려 사항: 새로 계산한 deltaTime의 최대값을 (예: 0.25초)로 설정하여 프로세스의 장시간 휴식(alt-tab등)에 대비해야 함

```
float t = 0.0f;
const float dt = 0.01f; //physics step
float currentTime = time();
float accumulator = 0.0f;
while (!quit) {
  float newTime = time();
  float deltaTime = newTime - currentTime; //display time delta
  currentTime = newTime;
  accumulator += deltaTime;
  while (accumulator >= dt) {
    integrate(state, t, dt);
    t += dt;
    accumulator -= dt;
  }
  render(state);
}
```

부드럽게 변하는 물리상태의 구현

- (두 framerate이 서로 정수배가 되지 않을 경우의) 문제점
  - 두 framerate가 유사하면,
    - 각 display update마다 physics step이 1번과 2번을 반복하게 됨
      → simulation이 부드럽지 못함
  - slow motion과 같은 oversampling의 경우에,
    - display framerate가 빨라도 physics update는 시간이 경과해야 이루어짐
      → simulation이 부드럽지 못함
- 해결책
  - 두 framerate가 서로 정수배가 아니면 차이가 생김 → 차이만큼 반영

```
float t = 0.0f;
const float dt = 0.01f;
float currentTime = time();
float accumulator = 0.0f;
while (!quit) {
  float newTime = time();
  float deltaTime = newTime - currentTime;
  currentTime = newTime;
  accumulator += deltaTime;
  while (accumulator >= dt) {
    integrate(state, t, dt);
    t += dt;
    accumulator -= dt;
  }
  if (accumulator > 0) {
    const float alpha = accumulator / dt; //alpha (0,1)
    State stateNext;
    integrate(stateNext, t+dt, dt);
    state = stateNext*alpha + state*(1.0f-alpha);
  }
  render(state);
}
```

예제 코드

- 적분법 예제 코드
- 적분법을 진자에 적용한 D3D 코드
- 충돌을 고려한 움직이는 다수 물체들 D3D 코드

01.PendulumConsole

02.PendulumBall

03.BouncingBalls