PHYSICS-BASED ANIMATION

CSA2207

Colin Vella

Animated Graphics

- Presentations
- Entertainment Media
- Simulations
- Computer Games

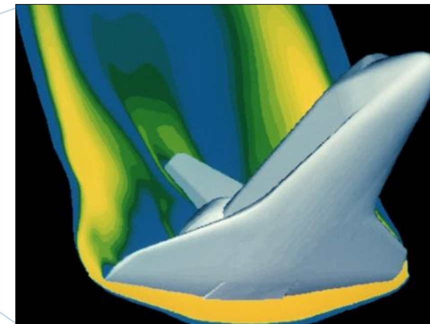



Digital Animation Approaches

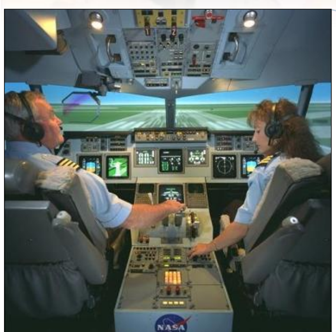
Scripted Animation	Interactive Animation
Ideal for predetermined sequences	Ideal for interactive applications
Requires prescription of the complete sequence	Requires a physical model and initial conditions
Can be designed for dramatic effect	Animation cannot be controlled directly
Requires skilled animators for realistic effects	Realism is a by-product of physics modelling
Animator resources / effort must scale in proportion to complexity	Computation resources must scale in proportion to complexity

Interactive Animation Applications

Engineering Design



Virtual Reality



Training Simulators

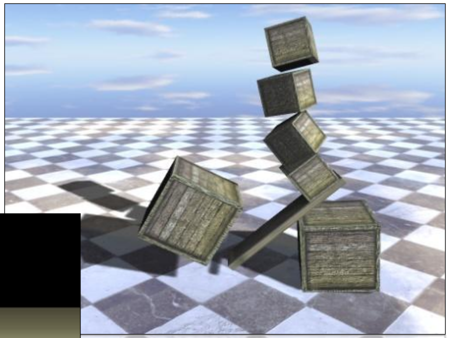
Computer Games

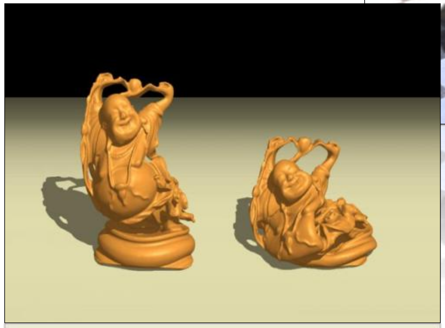
Existing Solutions

- Commercial / Closed Source
  - Havoc Physics™
  - Nvidia PhysX™
- Community Driven / Open Source
  - Bullet
  - Open Dynamics Engine™
  - Farseer

Physics Theory

- Classical Mechanics
  - Rigid Body Dynamics
  - Soft Body Dynamics

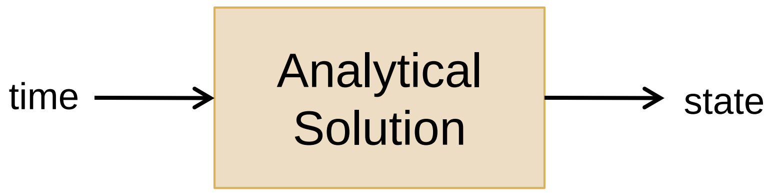




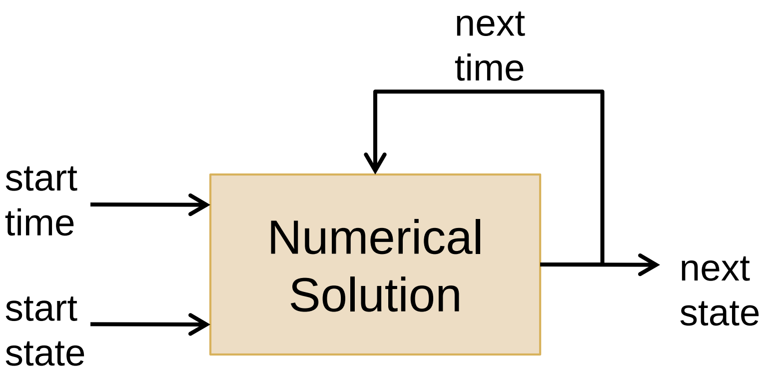
- Concepts
  - Linear and Angular Motion
  - Forces and Inertia
  - Collisions, Contact, Friction
  - Motion Constraints

Physical Models for Animation

Analytical Models



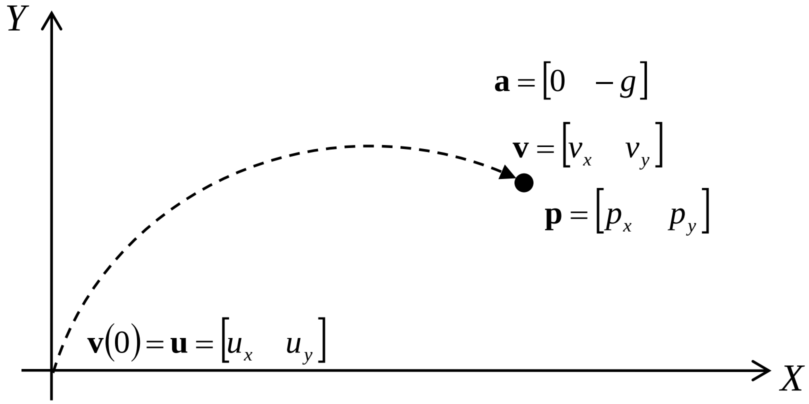
Numerical Models



Physical Models for Animation

Analytical Models	Numerical Models
Compute state as a function of time	Iteratively update state over small timeframes
Computationally efficient	Polynomial complexity
Very accurate (no error accumulation)	Numerical errors creep into simulations over time
Limited to simple predictable configurations with no interaction	Can handle interactive configurations of arbitrary complexity
Requires solution for every class of problem	Generic approach suitable to many problems

Analytical Model Example



Vector Equations of Motion

$$\mathbf{p}(t) = \mathbf{u}t + \frac{1}{2}\mathbf{a}t^2 \qquad \mathbf{v}(t) = \mathbf{u} + \mathbf{a}t$$

$$\mathbf{a}(t) = [0 \quad -g]$$

Component Equations of Motion

$$p_x(t) = u_x t \qquad v_x(t) = u_x \qquad a_x(t) = 0$$

$$p_y(t) = u_y t - \frac{1}{2}gt^2 \qquad v_y(t) = u_y - gt \qquad a_y(t) = -g$$

Algorithm

(1) Let t := 0

(2) Set initial velocity **u**

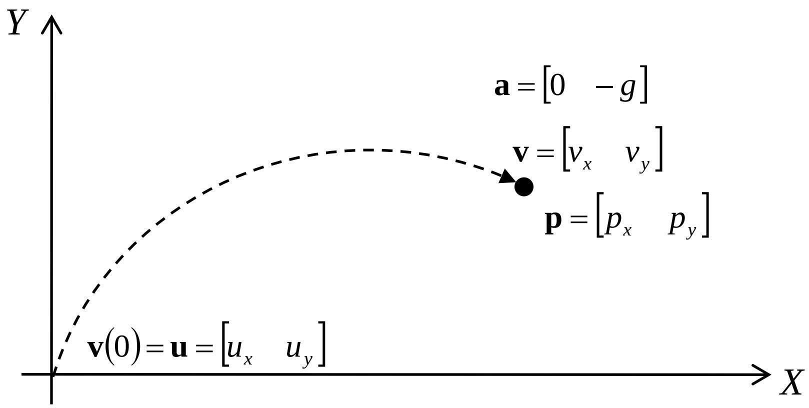
(3) Compute $\mathbf{p}(t)$

(4) Let $t := t + \Delta t$

(5) Draw projectile

(6) Go to step (3)

Analytical Model Example



Numerical Example

$\mathbf{u} = [3 \quad 4] \quad i.e. \quad u_x = 3 \quad u_y = 4$

$\mathbf{a} = [0 \quad -1] \quad i.e. \quad a_x = 0 \quad a_y = -1$

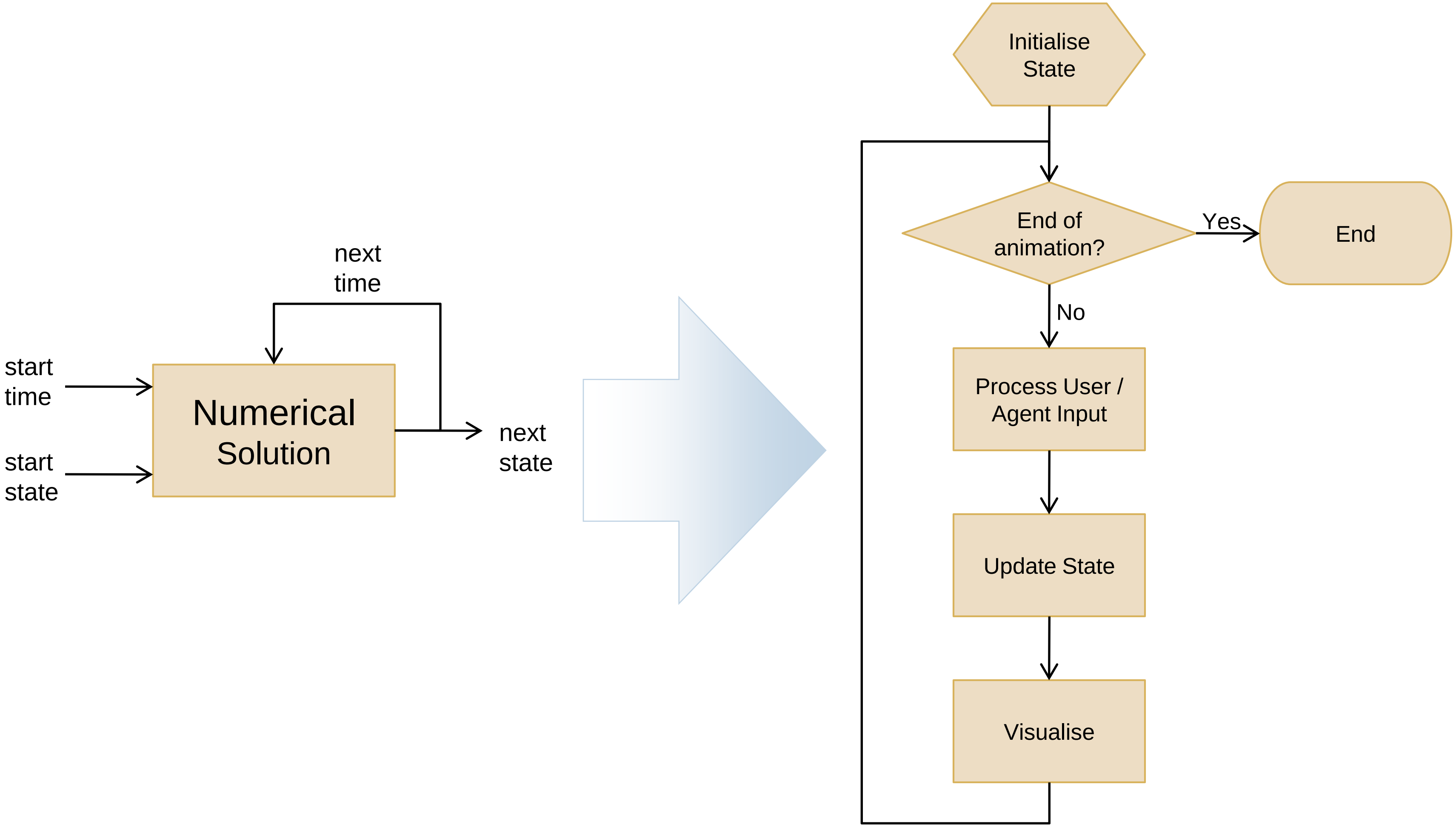
$\Delta t = 1$

$$p_x(t) = u_x t = 3t$$

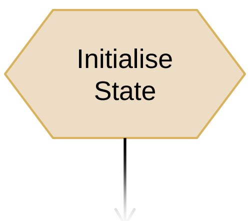
$$p_y(t) = u_y t + \frac{1}{2} a_y t^2 = 4t - \frac{1}{2} t^2$$

Time	$p_x=3t$	$p_y=4t-½t^2$	$p(t)=[p_x\ p_y]$
0	0.0	0.0	[0.0 0.0]
1	3.0	3.5	[3.0 3.5]
2	6.0	6.0	[6.0 6.0]
3	9.0	7.5	[9.0 7.5]
4	12.0	8.0	[12.0 8.0]
5	15.0	7.5	[15.0 7.5]
6	18.0	6.0	[18.0 6.0]
7	21.0	3.5	[21.0 3.5]
8	24.0	0.0	[24.0 0.0]

Numerical Animation Algorithm



State Initialisation



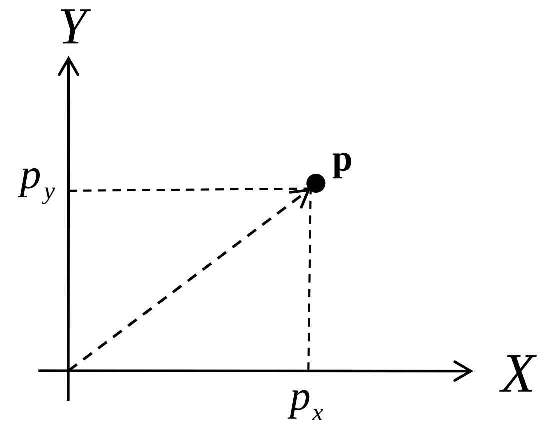
- What constitutes state?
- For each element (body)
  - Position
  - Orientation
- But also
  - Linear / Angular Velocity
  - Linear / Angular Acceleration
  - External Forces

(will deal with angular motion later...)

Representing Position

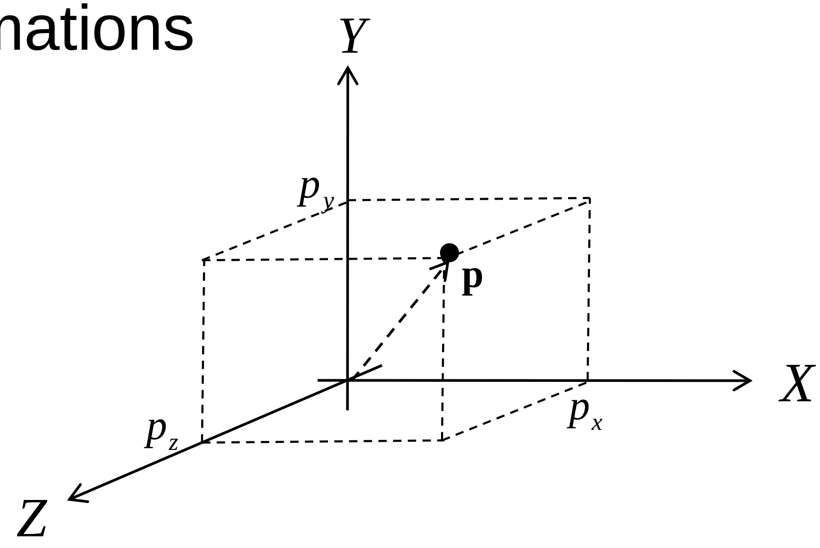
- Position Vectors
  - 2D Vectors for 2D Animations

$$\mathbf{p} = \begin{bmatrix} p_x & p_y \end{bmatrix}$$



  - 3D Vectors for 3D Animations

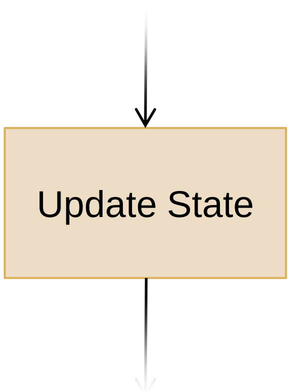
$$\mathbf{p} = \begin{bmatrix} p_x & p_y & p_z \end{bmatrix}$$



Representing Orientation

- Various representation options
- Bodies rotate around axis passing through a ‘central’ point (centre of mass)
- More on this later...

State Update



- For each body
  - Position changes due to linear velocity
  - Orientation changes due to angular velocity
  - Linear / angular velocity changes
    - due to linear / angular acceleration
    - due to some event, e.g. collision
- Linear / angular acceleration results from external forces

Representing Linear Velocity

- 2D or 3D Vectors

$$\mathbf{v} = \begin{bmatrix} v_x & v_y & v_z \end{bmatrix}$$

- Velocity is rate of change of position

$$\mathbf{v} = \frac{d\mathbf{p}}{dt} \qquad i.e. \qquad v_x = \frac{dp_x}{dt} \quad v_y = \frac{dp_y}{dt} \quad v_z = \frac{dp_z}{dt}$$

- i.e. integrating velocity over time gives position

$$\mathbf{p} = \mathbf{s} + \int_t \mathbf{v}dt \qquad i.e. \qquad p_x = s_x + \int_t v_x dt \quad p_y = s_y + \int_t v_y dt \quad p_z = s_z + \int_t v_z dt$$

- and if velocity constant, then

$$\mathbf{p} = \mathbf{s} + \mathbf{v}t \qquad i.e. \qquad p_x = s_x + v_x t \quad p_y = s_y + v_y t \quad p_z = s_z + v_z t$$

Representing Linear Acceleration

- 2D or 3D Vectors

$$\mathbf{a} = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix}$$

- Acceleration is rate of change of velocity

$$\mathbf{a} = \frac{d\mathbf{v}}{dt} \qquad i.e. \qquad a_x = \frac{dv_x}{dt} \quad a_y = \frac{dv_y}{dt} \quad a_z = \frac{dv_z}{dt}$$

- i.e. integrating acceleration over time gives velocity

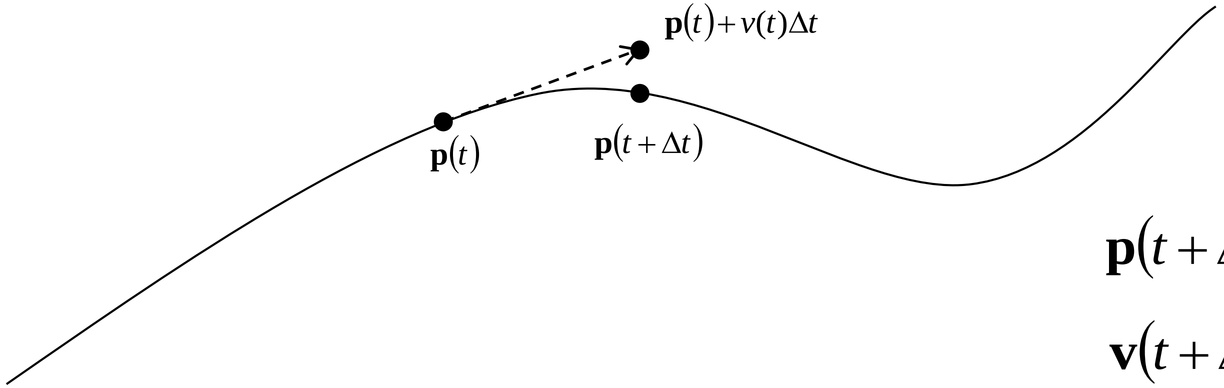
$$\mathbf{v} = \mathbf{u} + \int_t \mathbf{a}dt \qquad i.e. \qquad v_x = u_x + \int_t a_x dt \quad v_y = u_y + \int_t a_y dt \quad v_z = u_z + \int_t a_z dt$$

- and if acceleration constant, then

$$\mathbf{v} = \mathbf{u} + \mathbf{a}t \qquad i.e. \qquad v_x = u_x + a_x t \quad v_y = u_y + a_y t \quad v_z = u_z + a_z t$$

Numerical Integration

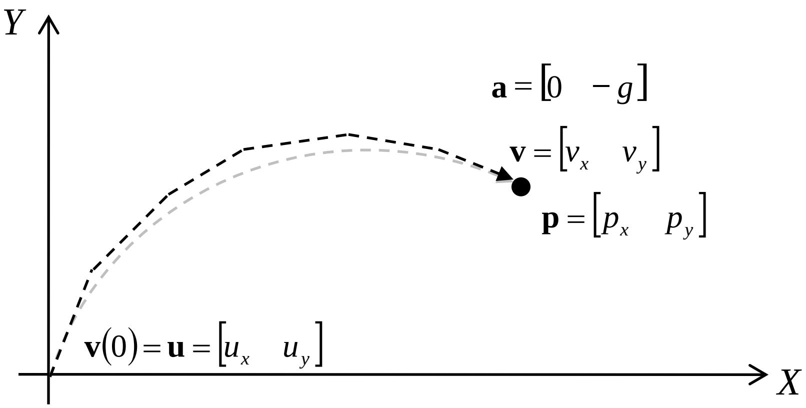
- Equations $\mathbf{p} = \mathbf{s} + \mathbf{v}t$ and $\mathbf{v} = \mathbf{u} + \mathbf{a}t$ valid only when **v** and **a** constant
- If **v** and **a** are variable, but t sufficiently small ($t = \Delta t$), we can use these equations to calculate approximations for $\mathbf{p}$ and $\mathbf{v}$
- We can calculate new value for $\mathbf{a}$ and repeat previous step
- This results in a *first order approximation* of the path taken by position $\mathbf{p}$



$$\mathbf{p}(t+\Delta t) \approx \mathbf{p}(t) + \mathbf{v}(t)\Delta t$$

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

Numerical Integration Example



Vector Equations of Motion

$$\mathbf{p}(0) = \mathbf{s} = [0 \quad 0] \qquad \mathbf{p}(t+\Delta t) \approx \mathbf{p}(t) + \mathbf{v}(t)\Delta t$$

$$\mathbf{v}(0) = \mathbf{u} \qquad \mathbf{v}(t+\Delta t) \approx \mathbf{v}(t) + \mathbf{a}(t)\Delta t$$

Component Equations of Motion

$$p_x(0) = \mathbf{s}_x = 0 \qquad p_x(t+\Delta t) \approx p_x(t) + v_x(t)\Delta t$$

$$p_y(0) = \mathbf{s}_y = 0 \qquad p_y(t+\Delta t) \approx p_y(t) + v_y(t)\Delta t$$

$$v_x = u_x$$

$$v_y(0) = u_y \qquad v_y(t+\Delta t) \approx v_y(t) - g\Delta t$$

Algorithm

(1) Let **p** := **s**, **v** := **u**, **a** := [0 -g]

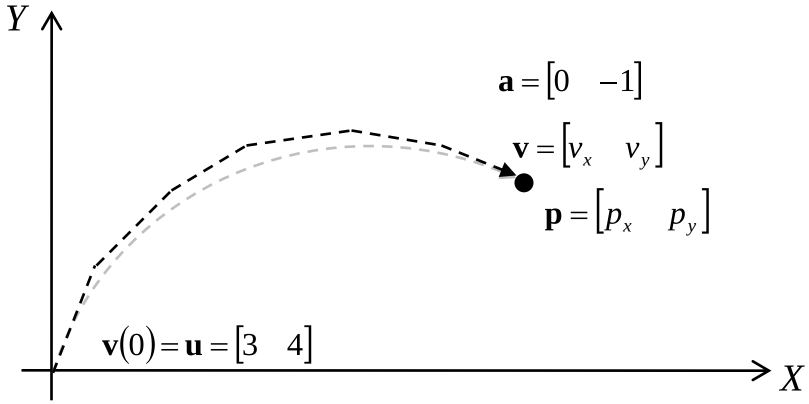
(2) Let **p'** := **p** + **v**Δ*t*, **v'** := **v** + **a**Δ*t*

(3) Let **p** := **p'**, **v** := **v'**

(4) Draw projectile

(5) Go to step (2)

Numerical Model Example



Numerical Example

$$\mathbf{p}(0) = [0 \quad 0]$$

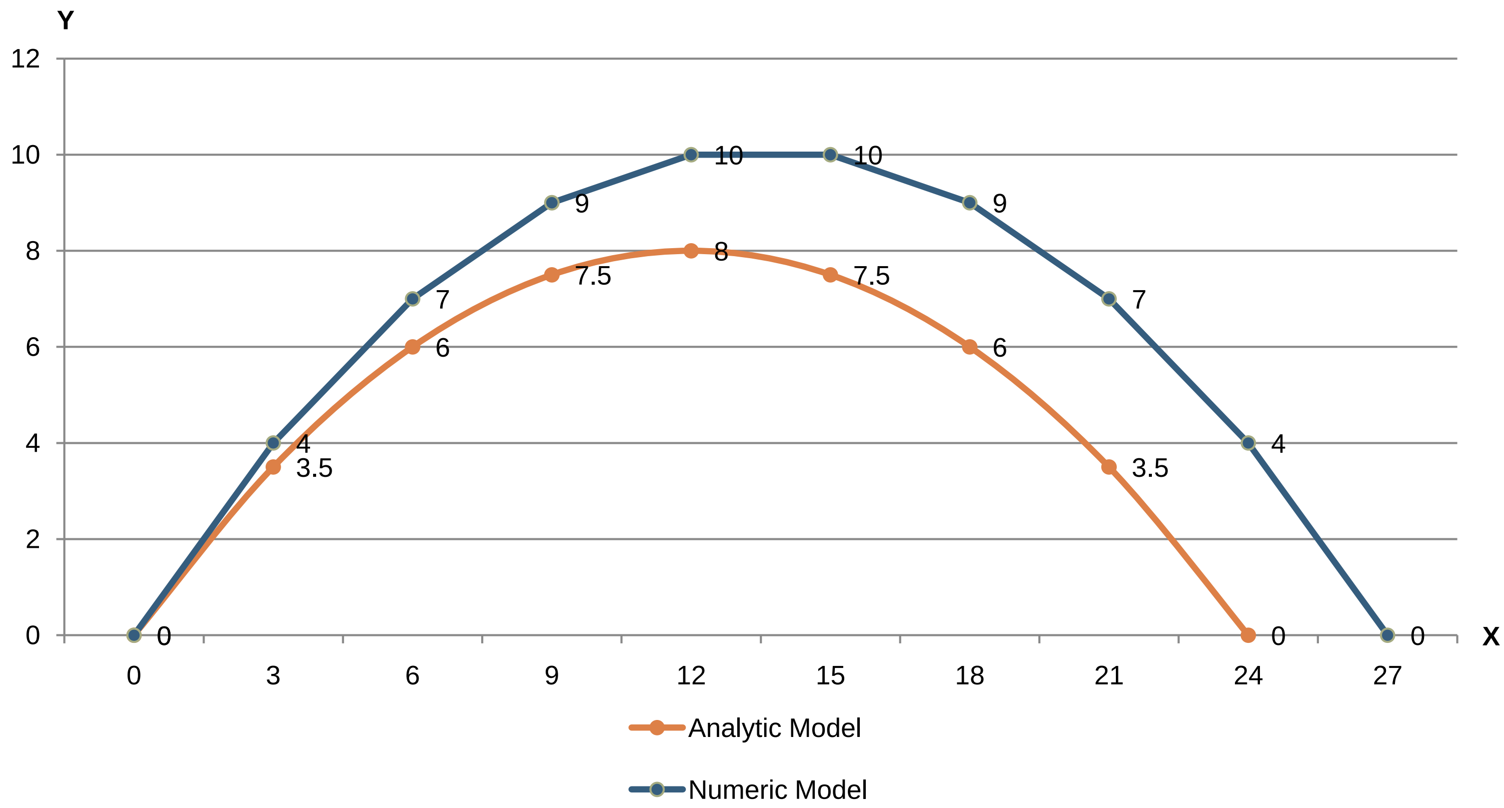
$$\mathbf{v}(0) = [3 \quad 4]$$

$$\mathbf{a} = [0 \quad -1]$$

$$t = 0, \quad \Delta t = 1$$

Time	p	v	a	p'=p+v	v'=v+a
0	[0 0]	[3 4]	[0 -1]	[3 4]	[3 3]
1	[3 4]	[3 3]	[0 -1]	[6 7]	[3 2]
2	[6 7]	[3 2]	[0 -1]	[9 9]	[3 1]
3	[9 9]	[3 1]	[0 -1]	[12 10]	[3 0]
4	[12 10]	[3 0]	[0 -1]	[15 10]	[3 -1]
5	[15 10]	[3 -1]	[0 -1]	[18 9]	[3 -2]
6	[18 9]	[3 -2]	[0 -1]	[21 7]	[3 -3]
7	[21 7]	[3 -3]	[0 -1]	[24 4]	[3 -4]
8	[24 4]	[3 -4]	[0 -1]	[27 0]	[3 -5]
9	[27 0]	[3 -5]	[0 -1]		

Analytic vs Numeric Results



Angular Motion

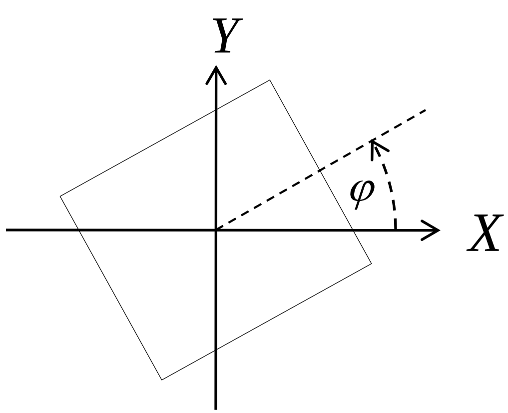
- We have angular equivalents of numerical equations for linear motion
  - $\boldsymbol{\varphi}$ is orientation
  - $\boldsymbol{\omega}$ is angular velocity
  - $\boldsymbol{\alpha}$ is angular acceleration

Linear Equations	Angular Equations
$\mathbf{p}(t+\Delta t) \approx \mathbf{p}(t) + \mathbf{v}(t)\Delta t$	$\boldsymbol{\varphi}(t+\Delta t) \approx \boldsymbol{\varphi}(t) + \boldsymbol{\omega}(t)\Delta t$
$\mathbf{v}(t+\Delta t) \approx \mathbf{v}(t) + \mathbf{a}(t)\Delta t$	$\boldsymbol{\omega}(t+\Delta t) \approx \boldsymbol{\omega}(t) + \boldsymbol{\alpha}(t)\Delta t$

2D Angular Motion

- Option 1: Scalar Angles
  - φ, ω, α expressed as scalars (in radians)
  - φ must be reduced to range [-π.. π] by adding / subtracting 2π

$$\varphi(t+\Delta t) \approx \varphi(t) + \omega(t)\Delta t \qquad \omega(t+\Delta t) \approx \omega(t) + \alpha(t)\Delta t$$



$$\omega = \frac{d\varphi}{dt} \qquad \alpha = \frac{d\omega}{dt}$$

2D Angular Motion

- Option 2: 2D Rotation Matrices
  - $\boldsymbol{\Phi}$ expressed as 2D rotation matrix
  - Angle of $\boldsymbol{\Phi}$ automatically falls within $[-\pi .. \pi]$

$$\Phi = \begin{bmatrix} \cos\varphi & -\sin\varphi \\ \sin\varphi & \cos\varphi \end{bmatrix}$$

  - ω, α still expressed as scalars
  - Must convert ω to rotation matrix to update $\boldsymbol{\varphi}$

$$\Phi(t+\Delta t) \approx \Phi(t) \begin{bmatrix} \cos\omega(t)\Delta t & -\sin\omega(t)\Delta t \\ \sin\omega(t)\Delta t & \cos\omega(t)\Delta t \end{bmatrix}$$

  - Angular velocity still updated as scalar

$$\omega(t+\Delta t) \approx \omega(t) + \alpha(t)\Delta t$$

  - $\boldsymbol{\varphi}$ loses orthogonality after a while, need renormalisation

2D Angular Motion

- Option 3: Complex Angles
  - $\boldsymbol{\varphi}$ expressed as complex number of unit length
  - Angle of $\boldsymbol{\varphi}$ automatically falls within $[-\pi .. \pi]$

$$\boldsymbol{\varphi} = e^{i\varphi\Delta t} = \cos\varphi\Delta t + i\sin\varphi\Delta t$$

  - ω, α still expressed as scalars
  - Must convert ω to complex number to update $\boldsymbol{\varphi}$

$$\boldsymbol{\varphi}(t+\Delta t) \approx \boldsymbol{\varphi}(t) e^{i\omega(t)\Delta t}$$

  - Angular velocity integration still computed as scalar

$$\omega(t+\Delta t) \approx \omega(t) + \alpha(t)\Delta t$$

  - May need to renormalise $\boldsymbol{\varphi}$ after a while

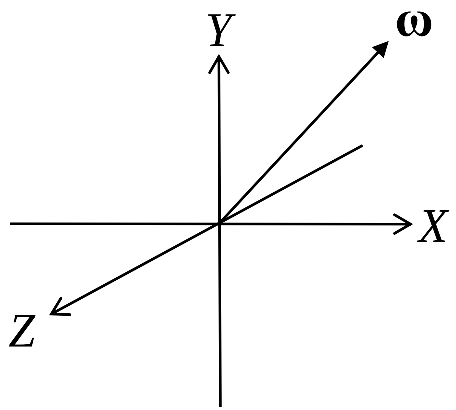
$$\boldsymbol{\varphi}' = \frac{1}{|\boldsymbol{\varphi}|}\boldsymbol{\varphi}$$

Comparison of 2D Rotation Structures

	Scalar Angles	2D Rotation Matrices	Complex Angles
Pros	• Very compact representation (1 scalar element) • Very cheap computation	• Solves angle discontinuity • Can reuse for visualisation	• Solves angle discontinuity • Compact representation (2 scalar elements) • Cheap ω conversion • Cheap conversion to matrix for visualisation • Cheap renormalisation
Cons	• Must handle angle discontinuity • Very costly conversion to matrix for visualisation	• Waste storage space (4 scalar elements • Expensive computations • Costly ω conversion • Costly renormalisation	• Less compact than scalar angles • Visualisation matrix still needs to be computed, but cheap

3D Angular Motion

- Option 1: Scaled Axis Representation
  - $\boldsymbol{\varphi}$, $\boldsymbol{\omega}$, $\boldsymbol{\alpha}$ expressed as vectors
    - Length represents scale of rotation
    - Direction represents axis of rotation
    - Rotation convention follows right-hand rule
  - Must reduce $|\boldsymbol{\varphi}|$ to range $[0..\ \pi]$ by subtracting 2π
  - Examples
    - $\boldsymbol{\varphi} = [0 \ \ 0 \ \ \pi/2\,]$ is a 90° anti-clockwise rotation around Z-axis
    - $\boldsymbol{\omega} = [4\pi \ \ 3\pi \ \ 0]$ is angular velocity of $5\pi/s$ around axis $y=3x/4$
    - $\boldsymbol{\alpha} = [2\pi \ \ 0 \ \ 0]$ is angular acceleration of $2\pi/s^2$ around axis X-axis



$$\boldsymbol{\varphi}(t+\Delta t) \approx \boldsymbol{\varphi}(t) + \boldsymbol{\omega}(t)\Delta t \qquad \boldsymbol{\omega}(t+\Delta t) \approx \boldsymbol{\omega}(t) + \boldsymbol{\alpha}(t)\Delta t$$

3D Angular Motion

- Option 2: 3D Rotation Matrices
  - $\boldsymbol{\Phi}$ expressed as 3D rotation matrix

$$\mathbf{\Phi} = \mathbf{R}_{\hat{\mathbf{n}},\varphi} = \begin{bmatrix} {n_x}^2 + \left(1 - {n_x}^2\right)c & n_x n_y (1-c) - n_z s & n_x n_z (1-c) + n_y s \\ n_x n_y (1-c) + n_z s & {n_y}^2 + \left(1 - {n_y}^2\right)c & n_y n_z (1-c) - n_x s \\ n_x n_z (1-c) - n_y s & n_y n_z (1-c) + n_x s & {n_z}^2 + \left(1 - {n_z}^2\right)c \end{bmatrix}$$

$$c = \cos\varphi$$
$$s = \sin\varphi$$

  - $\boldsymbol{\omega}$, $\boldsymbol{\alpha}$ still expressed as scaled axes representations
  - Must convert $\boldsymbol{\omega}$ to rotation matrix to update $\boldsymbol{\Phi}$

$$\mathbf{\Phi}(t + \Delta t) \approx \mathbf{R}_{\hat{\boldsymbol{\omega}},|\boldsymbol{\omega}|}\mathbf{\Phi}(t)$$

  - Angular velocity still updated as vector

$$\boldsymbol{\omega}(t + \Delta t) \approx \boldsymbol{\omega}(t) + \boldsymbol{\alpha}(t)\Delta t$$

  - $\boldsymbol{\Phi}$ loses orthogonality after a while, need renormalisation

3D Angular Motion

- Option 3: Quaternion Angles
  - About Quaternions
    - Like complex numbers, but in 4D
    - Have rules for addition, subtraction, multiplication etc.
  - Quaternions for Rotation
    - 3D equivalent of complex angles for 2D
    - Pros and cons analogous to complex numbers for 2D angular motion

Quaternions

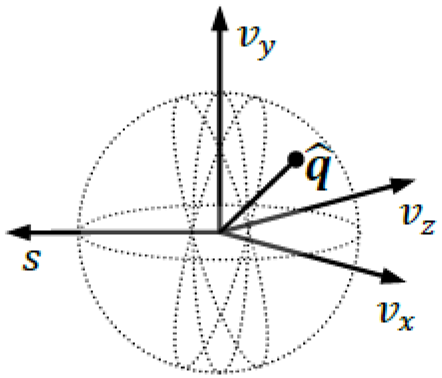
- 4D vectors with a special multiplicative operation
- Can be represented as a 4-element vector or a scalar / 3D vector pair

$$\mathbf{q} = \begin{bmatrix} s & \mathbf{v} \end{bmatrix} = \begin{bmatrix} s & v_x & v_y & v_z \end{bmatrix}$$

- Norm (Magnitude) $|\mathbf{q}| = \left|\begin{bmatrix} s & \mathbf{v} \end{bmatrix}\right| = \sqrt{s^2 + \mathbf{v}\cdot\mathbf{v}} = \sqrt{s^2 + v_x{}^2 + v_y{}^2 + v_z{}^2}$
- Conjugate $\mathbf{q}^* = \begin{bmatrix} s & \mathbf{v} \end{bmatrix}^* = \begin{bmatrix} s & -\mathbf{v} \end{bmatrix} = \begin{bmatrix} s & -v_x & -v_y & -v_z \end{bmatrix}$
- Multiplication $\mathbf{q}_1\mathbf{q}_2 = \begin{bmatrix} s_1 & \mathbf{v}_1 \end{bmatrix}\begin{bmatrix} s_2 & \mathbf{v}_2 \end{bmatrix} = \begin{bmatrix} s_1 s_2 - \mathbf{v}_1\cdot\mathbf{v}_2 & s_1\mathbf{v}_2 + s_2\mathbf{v}_1 + \mathbf{v}_1\times\mathbf{v}_2 \end{bmatrix}$
- Inverse $\mathbf{q}^{-1} = \dfrac{\mathbf{q}^*}{|\mathbf{q}|} = \dfrac{\begin{bmatrix} s & -\mathbf{v} \end{bmatrix}}{\sqrt{s^2 + \mathbf{v}\cdot\mathbf{v}}}$

Rotation Quaternions

- Unit quaternions can be used to rotate vectors
- Rotation by θ radians around unit vector **n**



$$\hat{\mathbf{q}} = \left[\cos\frac{\theta}{2} \quad \hat{\mathbf{n}}\sin\frac{\theta}{2}\right] \qquad\qquad |\hat{\mathbf{q}}| = 1$$

- Can rotate vector **v** to new vector **v'** as follows

$$[s' \quad \mathbf{v}'] = \hat{\mathbf{q}}[0 \quad \mathbf{v}]\hat{\mathbf{q}}^*$$

- Equation can be abbreviated for convenience

$$\mathbf{v}' = \hat{\mathbf{q}}\mathbf{v}\hat{\mathbf{q}}^*$$

Quaternion-Based Orientation

- Option 3: Quaternion Angles
  - $\boldsymbol{\varphi}$ expressed as a quaternion of unit norm
  - Angle of $\boldsymbol{\varphi}$ automatically falls within $[-\pi .. \pi]$

$$\boldsymbol{\varphi} = \mathbf{q}_{\hat{\mathbf{n}},\varphi} = \begin{bmatrix} \cos\frac{\varphi}{2} & \hat{\mathbf{n}}\sin\frac{\varphi}{2} \end{bmatrix}$$

  - $\boldsymbol{\omega}$, $\boldsymbol{\alpha}$ still expressed as scaled axis representations
  - Must wrap $\boldsymbol{\omega}$ in quaternion to update $\boldsymbol{\varphi}$

$$\boldsymbol{\varphi}(t+\Delta t) \approx \boldsymbol{\varphi}(t) + \frac{\Delta t}{2}\begin{bmatrix} 0 & \boldsymbol{\omega} \end{bmatrix}\boldsymbol{\varphi}(t)$$

  - Angular velocity integration still computed as scalar

$$\omega(t+\Delta t) \approx \omega(t) + \alpha(t)\Delta t$$

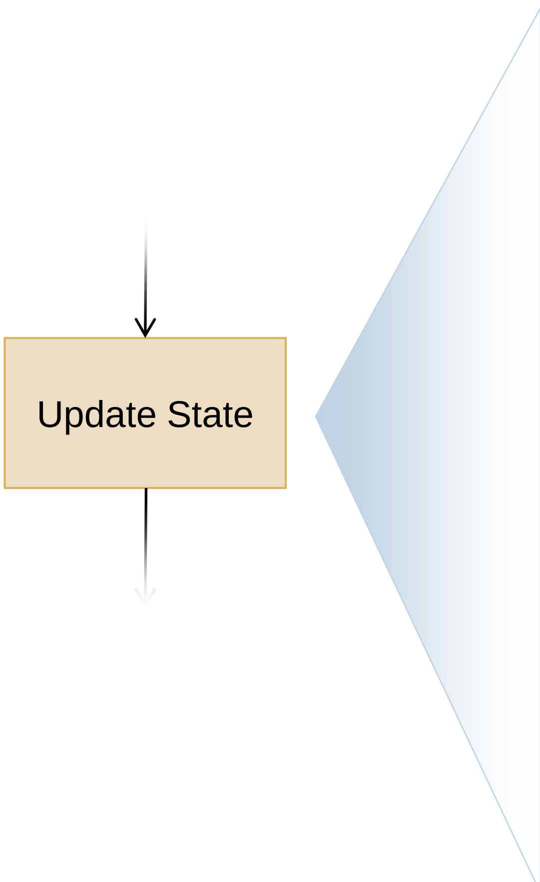
  - May need to renormalise φ after a while

$$\boldsymbol{\varphi}' = \frac{1}{|\boldsymbol{\varphi}|}\boldsymbol{\varphi}$$

Comparison of 3D Rotation Structures

	Scaled Axis Representations	3D Rotation Matrices	Quaternion Angles
Pros	• Very compact representation (3 scalar elements) • Very cheap computation	• Solves angle discontinuity • Can reuse for visualisation or cheaply convert to 4D homogenous matrix	• Solves angle discontinuity • Compact representation (4 scalar elements) • Cheap ω conversion • Reasonably cheap conversion to matrix for visualisation • Cheap renormalisation
Cons	• Must handle angle discontinuity • Very costly conversion to 3D/4D matrix for visualisation	• Wastes storage space (9 scalar elements • Expensive matrix computations • Costly ω conversion • Costly renormalisation	• Less compact than scaled axis representation • Visualisation matrix still needs to be computed, but relatively cheap

State Update (Take 2)



- For each body
  - Get current linear and angular acceleration (will tackle this next...)
  - Update position and orientation

$$\mathbf{p}(t+\Delta t) \approx \mathbf{p}(t) + \mathbf{v}(t)\Delta t$$

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

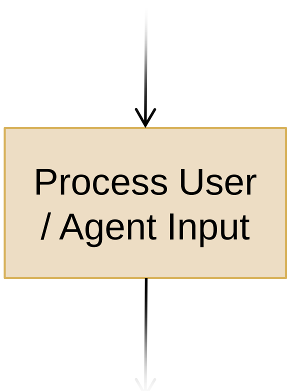
  - Update linear and angular velocities

$$\boldsymbol{\varphi}(t+\Delta t) \approx \boldsymbol{\varphi}(t) + \frac{\Delta t}{2}\begin{bmatrix}0 & \boldsymbol{\omega}\end{bmatrix}\boldsymbol{\varphi}(t)$$

$$\omega(t+\Delta t) \approx \omega(t) + \alpha(t)\Delta t$$

  - Handle collisions (will tackle this later...)

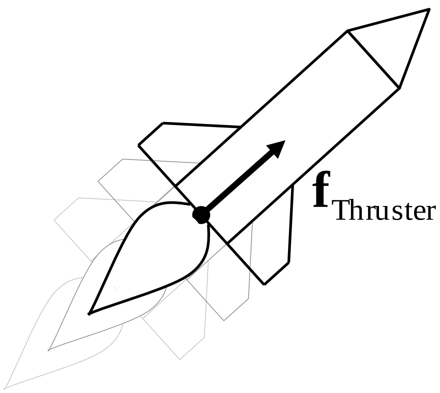
User / Agent Input

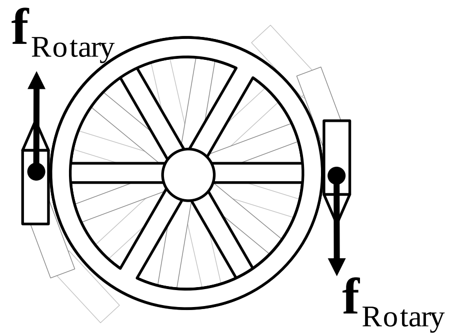


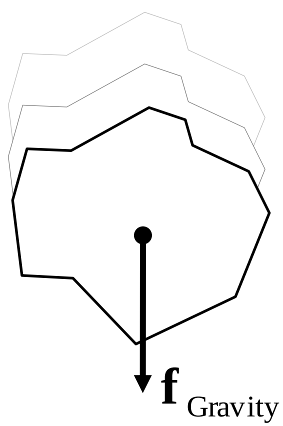
- Human users / autonomous agents influence physical simulation
- Examples
  - User / AI controlling simulated vehicle
  - Natural phenomena (e.g. gravity or friction)
  - Chain of events (e.g. collisions)
- The above result in applied forces
- Forces are source of linear and angular acceleration

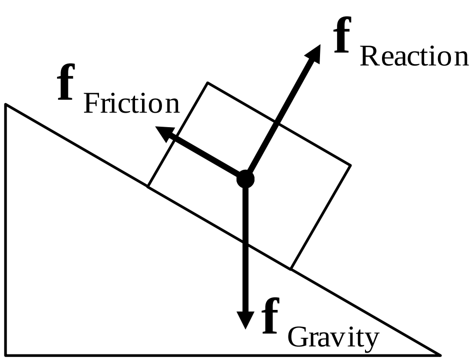
Force

- Has magnitude and direction (is a vector)
- Induce linear acceleration
- Induce angular acceleration (when acting off-centre)







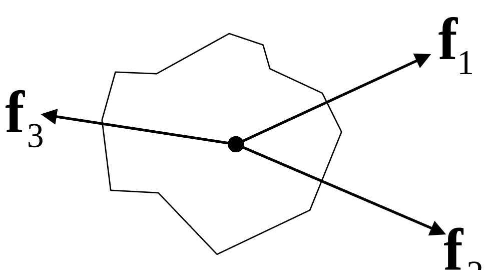


Effects of Force

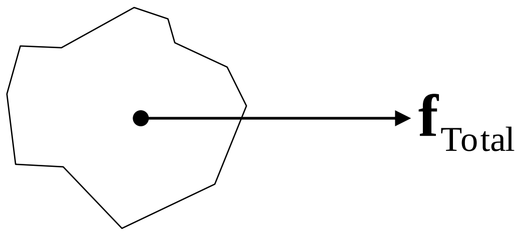
- Force induces linear acceleration
  - Greater force => greater acceleration
  - Greater mass => lesser acceleration
  - Acceleration parallel to force

$$\mathbf{f} = m\mathbf{a} \quad i.e. \quad \mathbf{a} = \frac{1}{m}\mathbf{f}$$

- Application of multiple forces
  - Forces can be summed up as vectors
  - Can work in tandem or cancel out



$$\mathbf{f}_{\text{Total}} = \sum_i \mathbf{f}_i$$



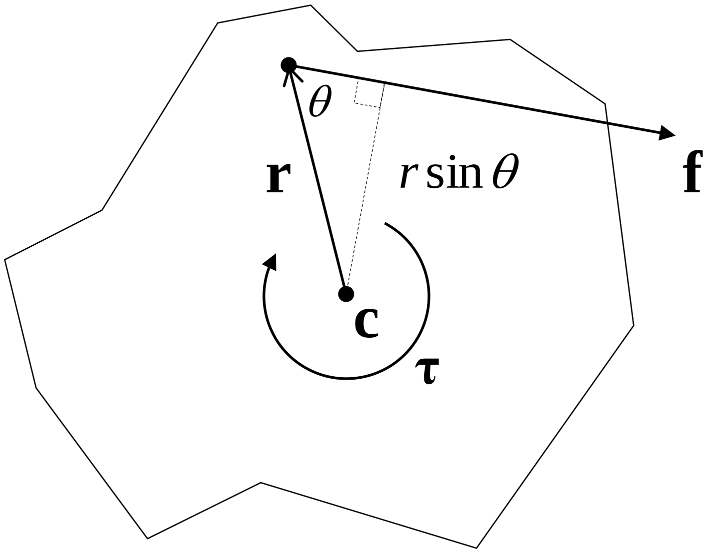
Torque

- Torque is ‘angular’ force
  - Magnitude of torque vector gives scale
  - Direction gives axis of rotation
  - greater force => greater torque
  - greater perpendicular distance => greater torque
- Scalar Form

$$\tau = \left(r \sin\theta\right) f$$

- Vector Form

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



Note: $\mathbf{c}$ is centre of mass

Effects of Torque

- Torque induces angular acceleration
  - Greater torque => greater acceleration
  - Greater 'mass' => lesser acceleration
  - Angular acceleration parallel to torque (for symmetric bodies)
  - Rotation occurs around axis passing through *centre of mass*

- Scalar Torque Equation

$$\tau = I\alpha \quad i.e. \quad \alpha = \frac{1}{I}\tau$$

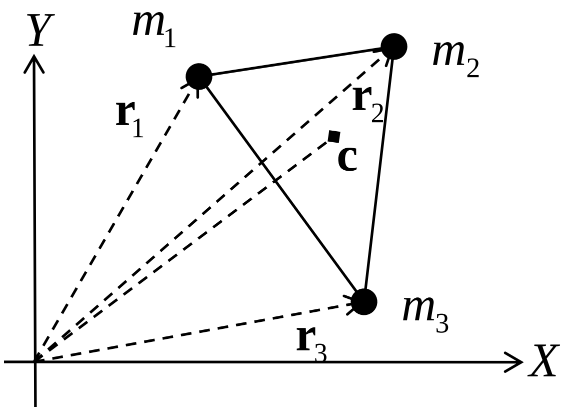
Note: Moment of Inertia (I) is angular equivalent of mass

Centre of Mass

- A point in (or outside) body around which mass is evenly distributed

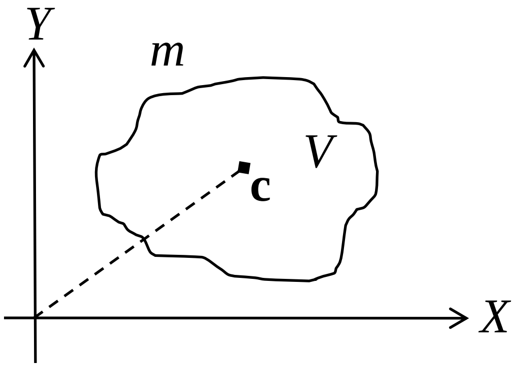
  - System of point masses m_i at positions $\mathbf{r}_i$

$$\mathbf{c} = \frac{\sum_i m_i \mathbf{r}_i}{\sum_i m_i}$$

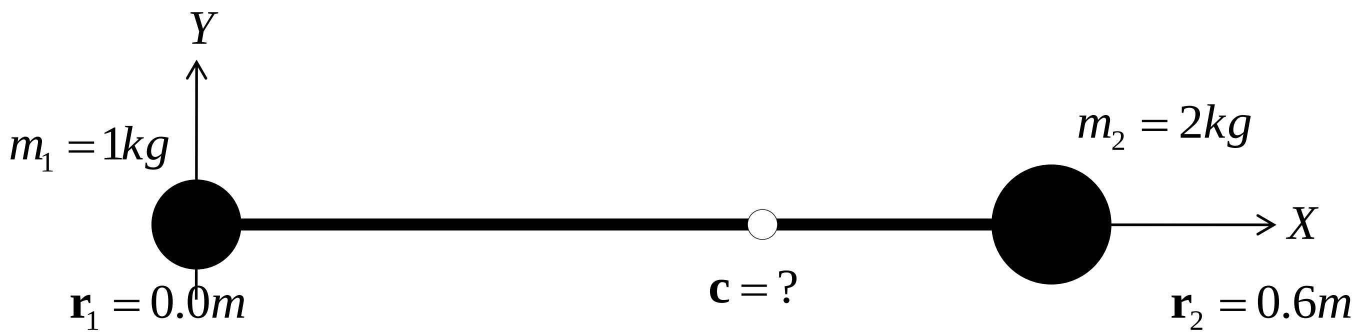


  - Continuous body mass m, density function ρ, volume V

$$\mathbf{c} = \frac{1}{m} \int_{\mathbf{r} \in V} \rho(\mathbf{r}) \mathbf{r} \, d\mathbf{r}$$



Centre of Mass Example



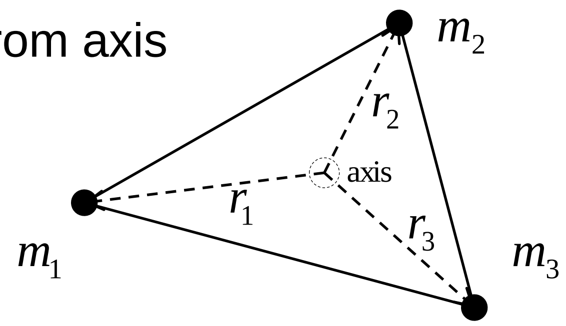
$$\mathbf{c} = \frac{m_1\mathbf{r}_1 + m_2\mathbf{r}_2}{m_1 + m_2} = \frac{1\times 0 + 2\times 0.6}{1+2} = \frac{1.2}{3} = 0.4m$$

Moment of Inertia

- A measure of mass quantity *and* distribution around a given axis (usually through centre of mass)

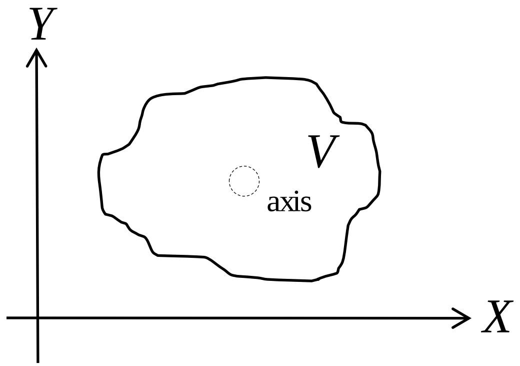
  - System of point masses m_i at perp. distance r_i from axis

$$I = \sum_i m_i r_i^2$$

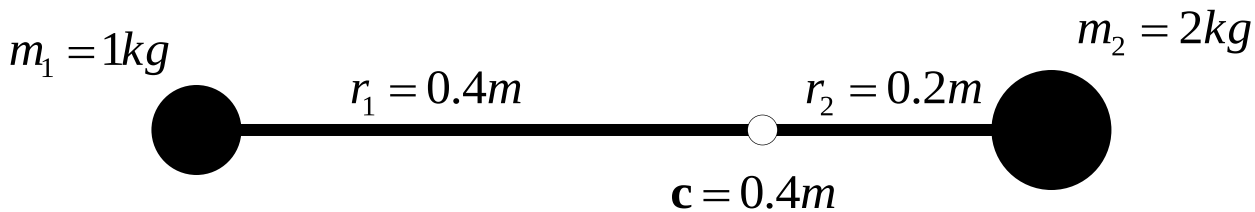


  - Solid body with density function ρ, volume V

$$I = \int_{\mathbf{r}\in V} \rho(\mathbf{r})\mathbf{r}^2 d\mathbf{r}$$



Moment of Inertia Example



$$\mathbf{c} = m_1\mathbf{r}_1{}^2 + m_2\mathbf{r}_2{}^2 = 1\times 0.2^2 + 2\times 0.4^2 = 0.36kgm^2$$

General Torque Equations

- For 2D, can use scalar forms of I, τ and α
- For 3D
  - Axis of rotation varies over time
  - Moment of inertia needs to be recalculated every time
  - Torque must take axis into account
  - Elegant Solution:
    - the *Inertia Tensor* matrix $\mathbf{I}$
    - vector form of the torque equations

$$\boldsymbol{\tau} = \mathbf{I}\boldsymbol{\alpha} \quad i.e. \quad \boldsymbol{\alpha} = \mathbf{I}^{-1}\boldsymbol{\tau}$$

$$\boldsymbol{\tau}_{\text{total}} = \sum_i \boldsymbol{\tau}_i = \sum_i \mathbf{r}_i \times \mathbf{f}_i$$

Moment of Inertia Tensor

- A 3 x 3 matrix of the form

$$\mathbf{I} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix}$$

- I_{xx}, I_{yy}, I_{zz} are *principal moments of inertia* around X, Y, Z axes

$$I_{xx} = \int_V \rho(\mathbf{r})\left(r_y{}^2 + r_z{}^2\right)dV \qquad I_{yy} = \int_V \rho(\mathbf{r})\left(r_x{}^2 + r_z{}^2\right)dV \qquad I_{zz} = \int_V \rho(\mathbf{r})\left(r_x{}^2 + r_y{}^2\right)dV$$

- I_{xy}, I_{xz}, I_{yx}, I_{yz}, I_{zx}, I_{zy} are *products of inertia*, usually zero for symmetrical bodies

$$I_{xy} = I_{yx} = -\int_V \rho(\mathbf{r}) r_x r_y dV \qquad I_{xz} = I_{zx} = -\int_V \rho(\mathbf{r}) r_x r_z dV$$

$$I_{yz} = I_{zy} = -\int_V \rho(\mathbf{r}) r_y r_z dV$$

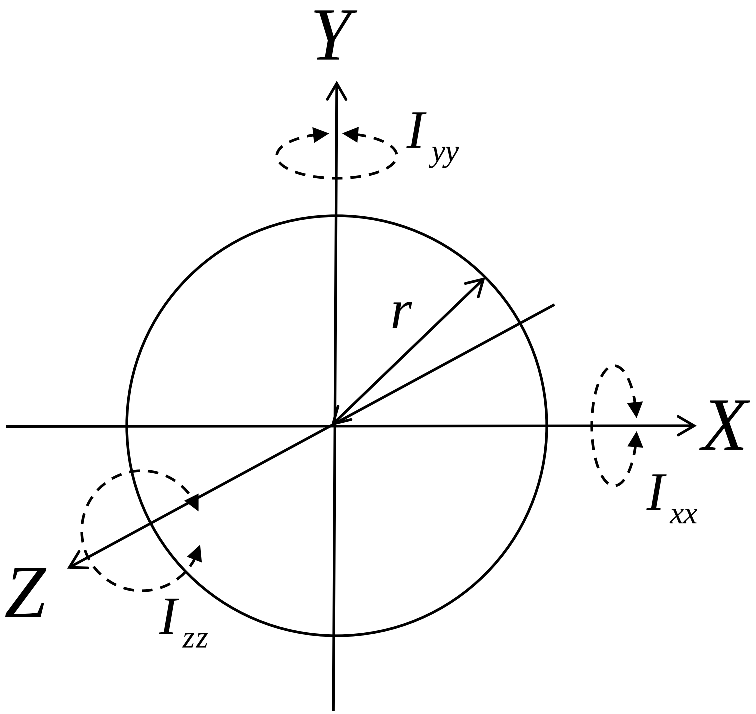
Inertia Tensor Example: Sphere

- Solid sphere of uniform density, mass m, radius r

$$I_{xx} = I_{yy} = I_{zz} = \frac{2}{5}mr^2$$

$$I_{xy} = I_{yx} = I_{xz} = I_{zx} = I_{yz} = I_{zy} = 0$$

$$\mathbf{I} = \begin{bmatrix} \frac{2}{5}mr^2 & 0 & 0 \\ 0 & \frac{2}{5}mr^2 & 0 \\ 0 & 0 & \frac{2}{5}mr^2 \end{bmatrix}$$



Inertia Tensor Example: Cuboid

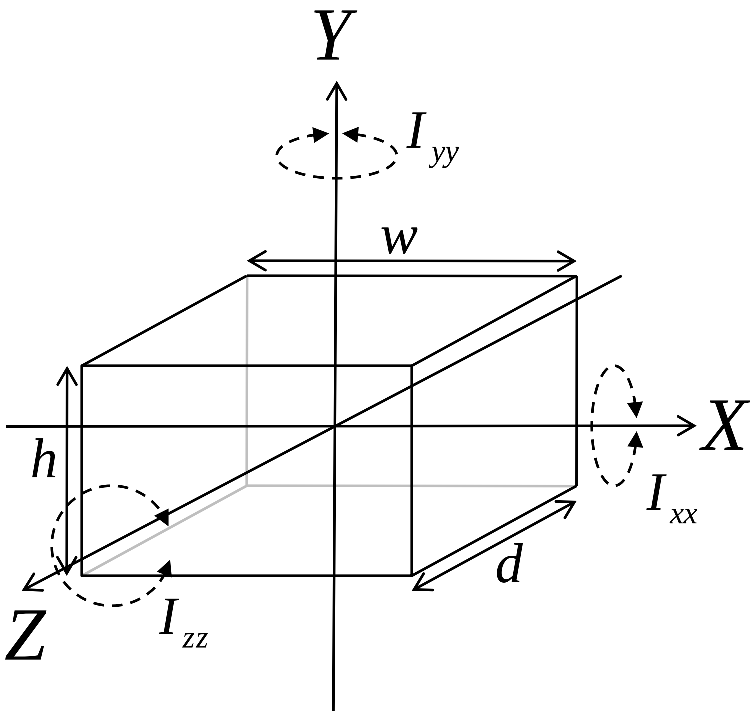
- Solid cuboid of uniform density, mass m, dimensions $w \times h \times d$

$$I_{xx} = \frac{m}{12}\left(h^2 + d^2\right) \qquad I_{yy} = \frac{m}{12}\left(w^2 + d^2\right)$$

$$I_{zz} = \frac{m}{12}\left(w^2 + h^2\right)$$

$$I_{xy} = I_{yx} = I_{xz} = I_{zx} = I_{yz} = I_{zy} = 0$$

$$\mathbf{I} = \begin{bmatrix} \frac{m}{12}\left(h^2 + d^2\right) & 0 & 0 \\ 0 & \frac{m}{12}\left(w^2 + d^2\right) & 0 \\ 0 & 0 & \frac{m}{12}\left(w^2 + h^2\right) \end{bmatrix}$$



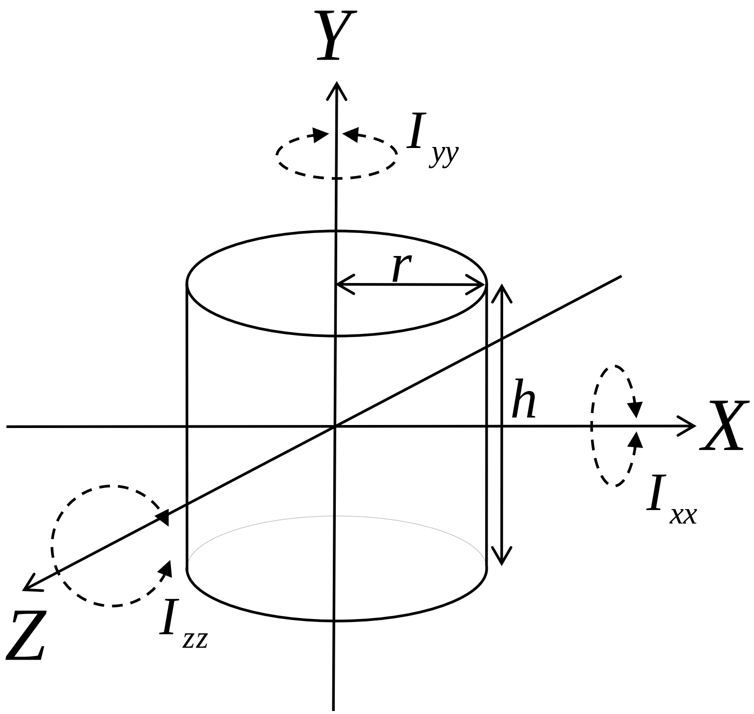
Inertia Tensor Example: Cylinder

- Solid cylinder of uniform density, mass m, height h, radius r

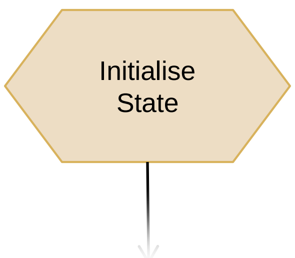
$$I_{xx} = I_{zz} = \frac{m}{12}\left(3r^2 + h^2\right) \qquad I_{yy} = \frac{mr^2}{2}$$

$$I_{xy} = I_{yx} = I_{xz} = I_{zx} = I_{yz} = I_{zy} = 0$$

$$\mathbf{I} = \begin{bmatrix} \frac{m}{12}\left(3r^2 + h^2\right) & 0 & 0 \\ 0 & \frac{mr^2}{2} & 0 \\ 0 & 0 & \frac{m}{12}\left(3r^2 + h^2\right) \end{bmatrix}$$

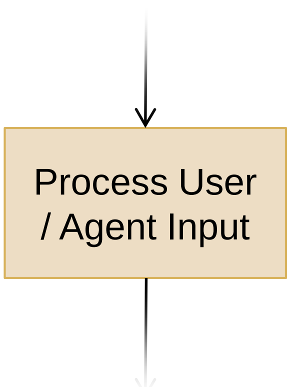


State Initialisation (Take 2)



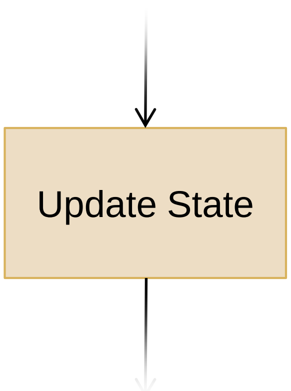
- For each body, initialise
  - Mass m
  - Moment of inertia tensor $\mathbf{I}$
  - Position vector $\mathbf{p}$
  - Orientation quaternion $\boldsymbol{\varphi}$
  - Linear velocity vector $\mathbf{v}$
  - Angular velocity vector $\boldsymbol{\omega}$

User / Agent Input (Take 2)



- For each body
  - Determine applied forces $\mathbf{f}_i$ from user / agent input
  - Accumulate force $$\mathbf{f}_{\text{Total}} = \sum_i \mathbf{f}_i$$
  - Determine torques $\boldsymbol{\tau}_i$ $$\boldsymbol{\tau}_i = \mathbf{r}_i \times \mathbf{f}_i$$
  - Accumulate torque $$\boldsymbol{\tau}_{\text{total}} = \sum_i \boldsymbol{\tau}_i$$

State Update (Take 3)



- For each body
  - Compute linear and angular accelerations

$$\mathbf{a} = \frac{1}{m}\mathbf{f}_{\text{total}} \qquad \boldsymbol{\alpha} = \mathbf{I}^{-1}\boldsymbol{\tau}_{\text{total}}$$

  - Update position and orientation

$$\mathbf{p}(t+\Delta t) \approx \mathbf{p}(t) + \mathbf{v}(t)\Delta t$$

$$\boldsymbol{\varphi}(t+\Delta t) \approx \boldsymbol{\varphi}(t) + \frac{\Delta t}{2}\left[0 \quad \boldsymbol{\omega}\right]\boldsymbol{\varphi}(t)$$

  - Update linear and angular velocities

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

$$\omega(t+\Delta t) \approx \omega(t) + \alpha(t)\Delta t$$

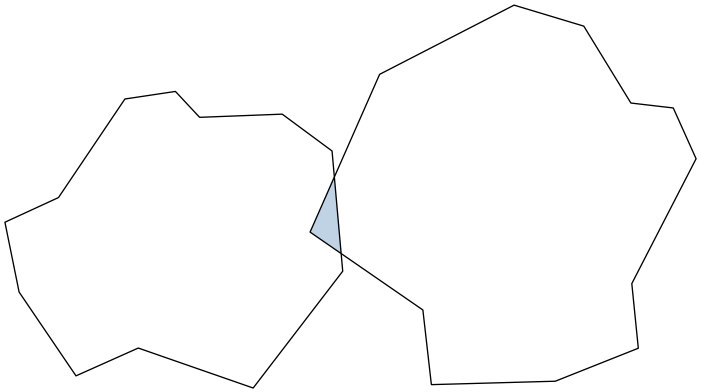
Collision Detection and Response

- Need to prevent bodies from interpenetrating
- Need to maintain realism

- Two problems:
  - How to detect a collision?
  - What to do when a collision occurs?

Collision Detection

- Bodies occupy volume in space
- Collision occurs when volumes overlap on at least one point in space



- Two possible approaches
  - *Conservative Advancement*: Estimate time of collision before it occurs
  - *Retroactive Detection*: Let bodies overlap and fix penetration afterwards

Conservative Advancement

- In current state update
  - For all possible collisions, estimate time of impact Δt_{impact} (less than usual update interval Δt)
  - If there is such collision
    - update motion equations by Δt_{impact} (instead of Δt)
    - handle collision (e.g. update velocities)
    - resume normally
  - Otherwise if no collision
    - Update motion equations by Δt as usual

- Problems of this approach
  - Time of impact estimation is harder than testing if bodies overlap
  - Simulation comes to virtual stop when lots of bodies in contact
  - More difficult to keep constant animation rate

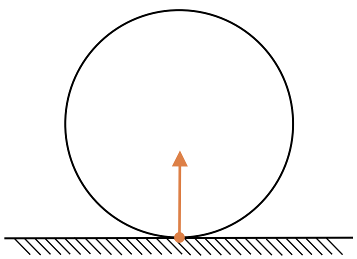
Retroactive Collision Detection

- In current state update
  - Update motion of all bodies by Δt
  - For each overlapping pair of bodies
    - Fix penetration (e.g. back off bodies to earlier position)
    - Handle collision (e.g. update velocities)

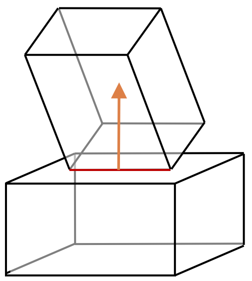
- Problems with this approach
  - Must deal with interpenetration
  - Tunnelling problem (small bodies, high velocities, large Δt)
  - Stacking problem (will talk about this later...)

Collision Manifolds

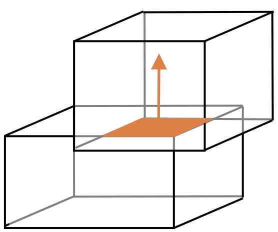
- Area of contact (manifold) between colliding bodies can be
  - a single point
  - a discreet number of points
  - a continuum of points (line / area)
  - a mix of the above
- Common occurrences
  - corner with side (vertex – face)
  - edge with edge (edge – edge)
  - edge with surface (edge – face)
- Other types (rare)
  - corner with corner
  - corner with edge
- Lines / areas of contacts simplified to discreet points



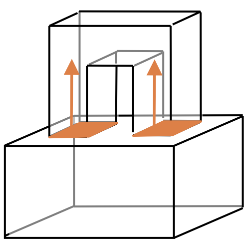
point of contact



line of contact



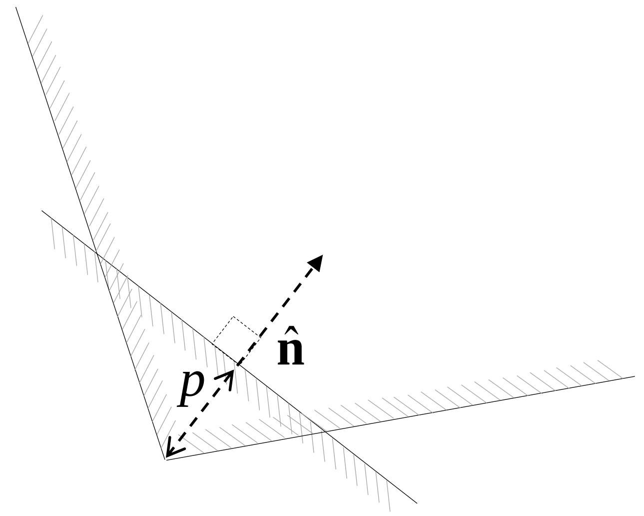
area of contact



multiple areas of contact

Collision Detection Output

- For each discreet point of collision we need
  - Point of contact
    - Location where collision has occurred
  - Contact normal vector **ň**
    - Direction of the collision
  - Penetration distance p
    - For resolving interpenetration



Sphere Collision Detection Example

- Sphere 1, centre at $\mathbf{p}_1$, radius r_1
- Sphere 2, centre at $\mathbf{p}_2$, radius r_2
- Spheres in contact / overlapping when

$$\left|\mathbf{p}_2 - \mathbf{p}_1\right| = d \leq r_1 + r_2$$

- If overlapping, then penetration p is

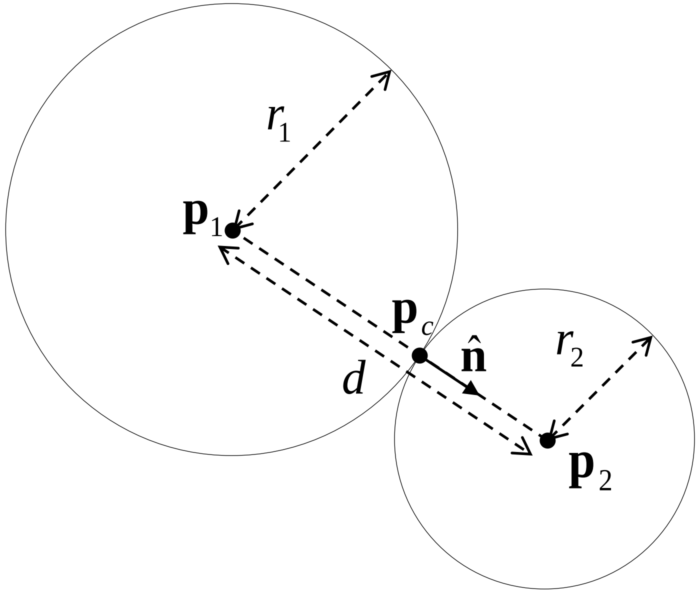
$$p = r_1 + r_2 - d$$

- Contact normal ň is

$$\hat{\mathbf{n}} = \frac{1}{d}\left(\mathbf{p}_2 - \mathbf{p}_1\right)$$

- Point of contact $\mathbf{p}_c$ is (approximately)

$$\mathbf{p}_c = \mathbf{p}_1 + \left(r_1 - \frac{p}{2}\right)\hat{\mathbf{n}}$$



Collision Detection Performance

- Simplest solution: test all possible body pairs – $n(n\text{-}1)/2$ combinations!

- Better approaches: partition space for better performance, for example:
  - Regular grids
  - Quadtrees (2D) / octrees (3D)
  - KD-trees
  - co-ordinate sorting

Regular Grids

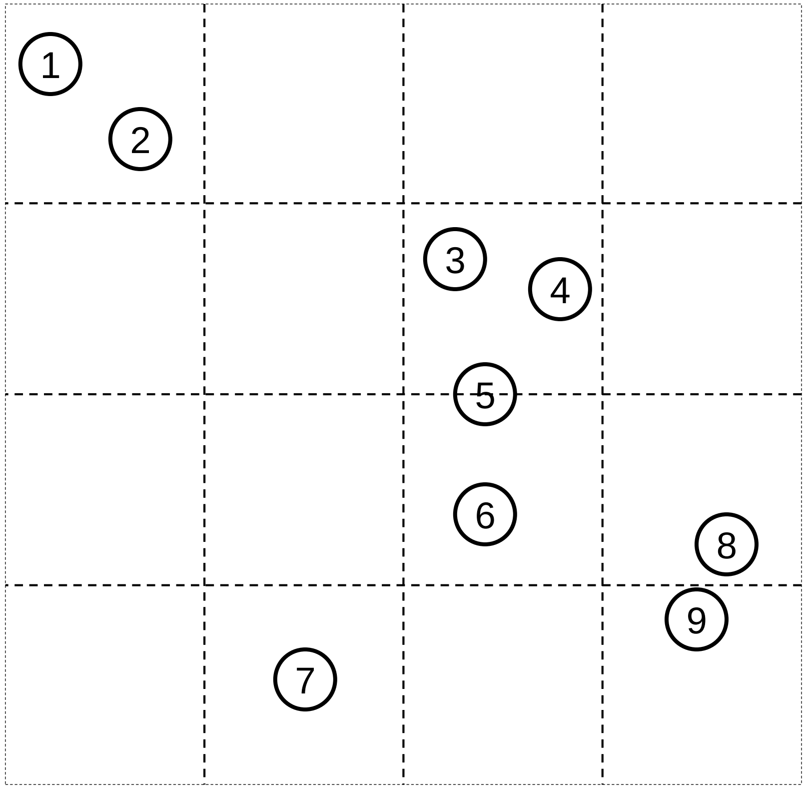
- Test only bodies sharing same cells

- Example, test only the following
  - (1) and (2)

  - (3) and (4)
  - (3) and (5)
  - (4) and (5)

  - (5) and (6) – body (5) spans 2 cells

  - **Note:** (7), (8), (9) ignored



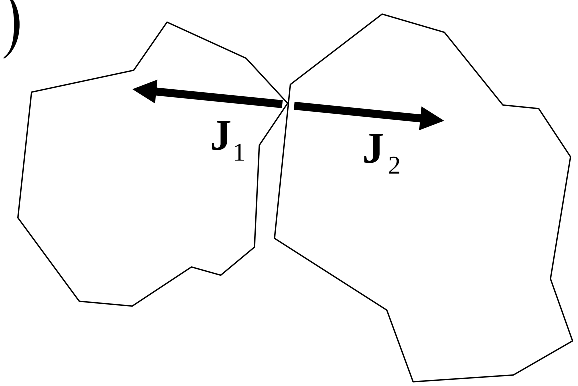
- Only 5 out of 36 possible combinations tested!

Collision Response

- In a real collision
  - Bodies undergo compression, followed by expansion before breaking contact, over short period of time
  - During compression and expansion phases, repulsive forces (along contact normal) accelerate bodies apart
  - Linear and angular velocities change gradually throughout collision
- In a simulated collision between perfectly rigid bodies
  - We avoid simulating compression and expansion phases
  - We model repulsive force by instantaneous change in momentum (*impulse*)

$$\mathbf{J}_1 = m_1\left(\mathbf{v}'_{n1} - \mathbf{v}_{n1}\right) = -\mathbf{J}_2 = m_2\left(\mathbf{v}'_{n2} - \mathbf{v}_{n2}\right)$$

  - Linear and angular velocities change instantly



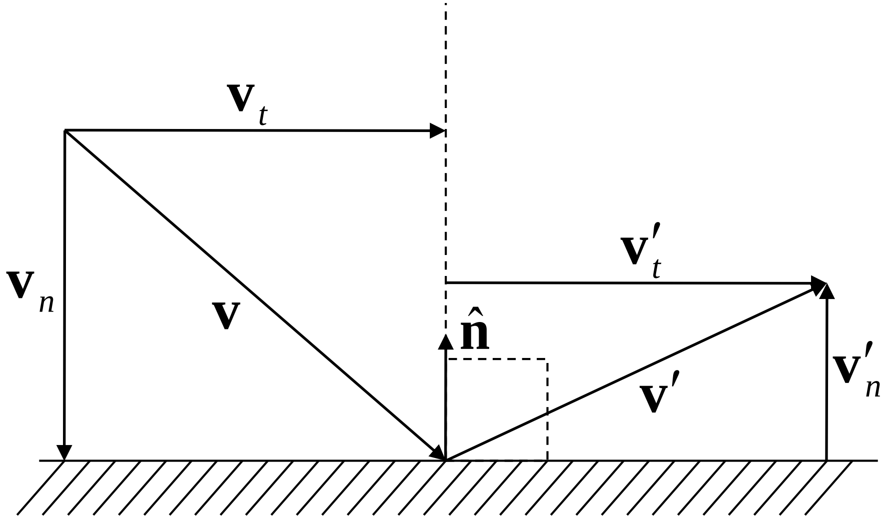
Coefficient of Restitution

- In a frictionless rigid body collision, relative velocity of contact points
  - changes only along contact normal
  - is unaffected along perpendicular direction to normal (surface tangent)

- Collision modelled by restitution coefficient e with value between 0 and 1
  - $e = 1$ => perfectly elastic collision
  - $e = 0$ => perfectly inelastic (sticky) collision
  - measured empirically e.g. wooden ball hitting concrete $e \approx 0.6$

$$e = \frac{|\mathbf{v}'_n|}{|\mathbf{v}_n|}$$

$$\mathbf{v}' = \mathbf{v} - (\mathbf{v} \cdot \hat{\mathbf{n}}(1+e))\hat{\mathbf{n}}$$



Collision Effects

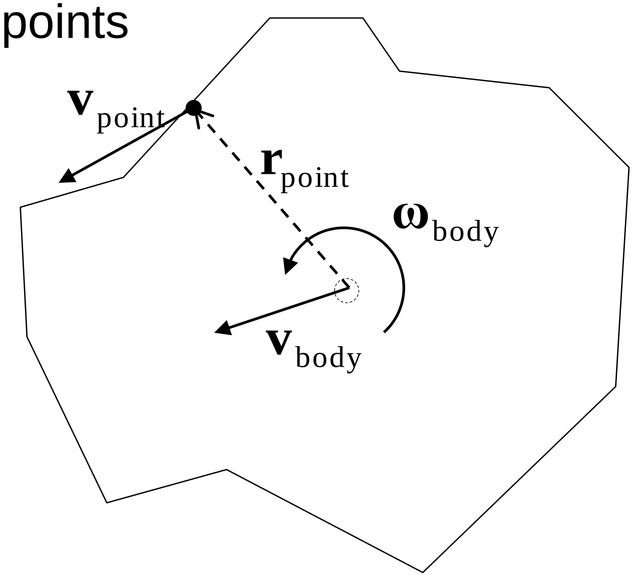
- Relative velocity of contact points changes according to coefficient e (as per previous slide)
- Can compute contact point velocity from linear and angular body velocity

$$\mathbf{v}_{\text{contact}} = \mathbf{v}_{\text{body}} + \mathbf{r}_{\text{contact}} \times \boldsymbol{\omega}_{\text{body}}$$

- Then compute relative velocity of contact points

$$\mathbf{v}_r = \mathbf{v}_{\text{contact2}} - \mathbf{v}_{\text{contact1}}$$

- Several substitutions later lead to...



Collision Equation

- Step 1: Computation of impulse magnitude j

$$j = \frac{-(1+e)\mathbf{v}_r \cdot \hat{\mathbf{n}}}{\frac{1}{m_1} + \frac{1}{m_2} + \left(I_1^{-1}(\mathbf{r}_1 \times \hat{\mathbf{n}}) \times \mathbf{r}_1 + I_2^{-1}(\mathbf{r}_2 \times \hat{\mathbf{n}}) \times \mathbf{r}_2\right) \cdot \hat{\mathbf{n}}}$$

- Step 2: Vector forms of impulses $\mathbf{j}_1$, $\mathbf{j}_2$

$$\mathbf{j}_1 = j\hat{\mathbf{n}} \qquad \mathbf{j}_2 = -j\hat{\mathbf{n}}$$

- Step 3a: New linear velocities $\mathbf{v}'_1$, $\mathbf{v}'_2$

$$\mathbf{v}'_1 = \mathbf{v}_1 + \frac{1}{m_1}\mathbf{j}_1 \qquad \mathbf{v}'_2 = \mathbf{v}_2 + \frac{1}{m_2}\mathbf{j}_2$$

- Step 3b: New angular velocities $\boldsymbol{\omega}'_1$, $\boldsymbol{\omega}'_2$

$$\boldsymbol{\omega}'_1 = \boldsymbol{\omega}_1 + \mathbf{I}_1^{-1}(\mathbf{r}_1 \times \mathbf{j}_1) \qquad \boldsymbol{\omega}'_2 = \boldsymbol{\omega}_2 + \mathbf{I}_2^{-1}(\mathbf{r}_2 \times \mathbf{j}_2)$$

Solving Interpenetration

- Option 1 (Simple)
  - Move each body away by half penetration along contact normal

$$\mathbf{p}_1' = \mathbf{p}_1 - \frac{p}{2}\hat{\mathbf{n}} \qquad \mathbf{p}_2' = \mathbf{p}_2 + \frac{p}{2}\hat{\mathbf{n}}$$

- Option 2 (Better)
  - Move each body away taking mass into consideration

$$\mathbf{p}_1' = \mathbf{p}_1 - \frac{m_2}{m_1 + m_2} p\hat{\mathbf{n}} \qquad \mathbf{p}_2' = \mathbf{p}_2 + \frac{m_1}{m_1 + m_2} p\hat{\mathbf{n}}$$

- Option 3 (Even Better)
  - Apply ‘impulse’ equation at positional level (handles rotation)

Collision Algorithm

- For each collision

  (1) Compute collision impulse

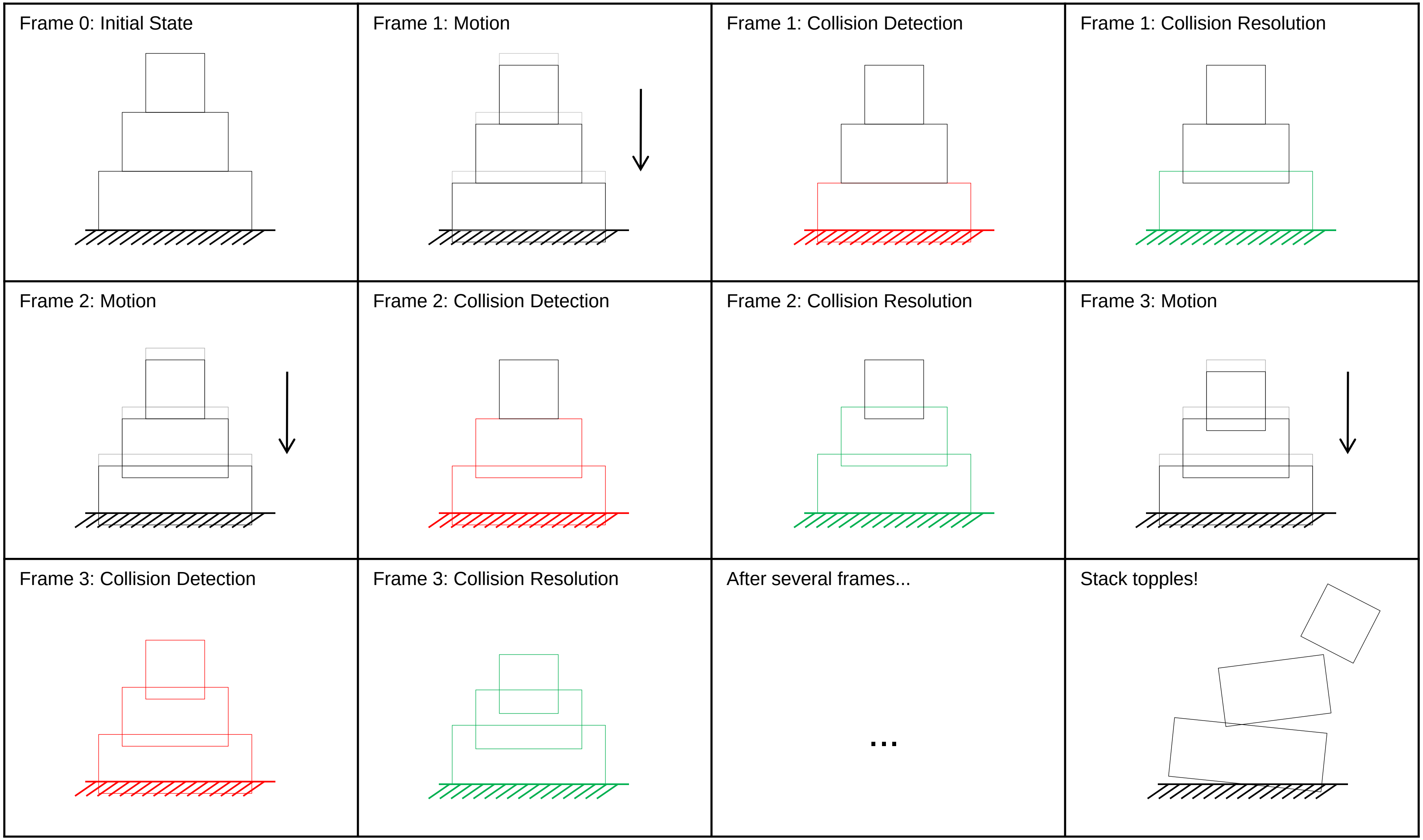
  (2) Update linear velocities

  (3) Update angular velocities

  (4) Solve body interpenetration

- Problems
  - Solving one interpenetration may cause another
  - Cannot handle stacks of bodies

The Stacking Problem



Simultaneous Collision Resolution

- All collisions considered simultaneously
- Solves (or minimises) stacking problem

- Various solutions (look up for fun...)
  - Shock Propagation
  - Iterative Solver
  - Linear Complementary Problem Formulation

Further Topics on Physics Animation

- Simulating friction, for example:
  - Static box on inclined plane
  - Tyre traction

- Joints, for example:
  - Ball-and-socket
  - Hinges
  - Motors

- Modelling Forces, for example:
  - Springs
  - Buoyancy

Some References

- Physics Engines
  - http://en.wikipedia.org/wiki/Physics_engine
- Collision Detection
  - http://en.wikipedia.org/wiki/Collision_detection
- Collision Response
  - http://en.wikipedia.org/wiki/Collision_response
- List of Inertia Tensors
  - http://en.wikipedia.org/wiki/List_of_moment_of_inertia_tensors
- Octrees
  - http://en.wikipedia.org/wiki/Octree
- Open Source / Free Physics Engines
  - http://www.thefreecountry.com/sourcecode/physics.shtml
- Farseer Physics Engine (XNA Friendly)
  - http://www.farseergames.com/storage/farseerphysics/Manual2.1.htm