From Particles to Rigid Bodies

- **Particles**
  - No rotations
  - Linear velocity $v$ only
  - 3N DoFs

- **Rigid bodies**
  - 6 DoFs (translation + rotation)
  - Linear velocity $v$
  - Angular velocity $\omega$
Outline

- Rigid Body Representation
- Kinematics
- Dynamics
- Simulation Algorithm
- Collisions and Contact Response
Coordinate Systems

• Body Space (Local Coordinate System)
  – Rigid bodies are defined relative to this system
  – Center of mass is the origin (for convenience)
  • We will specify body-related physical properties (inertia, ...) in this frame
Coordinate Systems

- World Space:
  rigid body transformation to common frame

\[ \mathbf{p}(t) = \mathbf{x}(t) + \text{Rot}(\mathbf{p}_0) \]

- Translation
- Rotation
Center of mass

• **Definition**

\[ x_0 = \frac{\sum m_i x_i}{\sum m_i} = \frac{\sum m_i x_i}{M} \]

\[ M x_0 = \sum m_i x_i \]

• **Motivation: forces**

(one mass particle:)

\[ f_i = m_i \ddot{x}_i \]

(entire body:)

\[ F = \sum f_i = \frac{d^2}{dt^2} \sum m_i x_i \]

\[ F = M \ddot{x}_0 \]
Rotations

- Euler angles:
  - 3 DoFs: roll, pitch, heading
  - Dependent on order of application
  - Not practical
Rotations

- Rotation matrix
  - 3x3 matrix: 9 DoFs
  - Columns: world-space coordinates of body-space base vectors
  - Rotate a vector: $\text{Rot}(v) = Rv = \begin{pmatrix} a_1 & a_1 & a_1 \end{pmatrix} v$
Rotations

• Problem with rotation matrices: numerical drift

\[ R(t_k) = \Delta t^k \dot{R}(t_k) \dot{R}(t_{k-1}) \dot{R}(t_{k-2}) \ldots R(t_0) \]

• Fix: use Gram-Schmidt orthogonalization
• Drift is easier to fix with quaternions
Unit Quaternion Definition

- \( q = [s, v] : s \) is a scalar, \( v \) is vector
- A rotation of \( \theta \) about a unit axis \( u \) can be represented by the unit quaternion:
  \[
  [\cos(\theta/2), \sin(\theta/2) \cdot u]
  \]
- Rotate a vector: \( \text{Rot}(v) = qaq^* \)
- Fix drift:
  - 4-tuple: vector representation of rotation
  - Normalized quaternion always defines a rotation in \( \mathbb{R}^3 \)
Unit Quaternion Operations

• Special multiplication:
\[
[s_1, v_1][s_2, v_2] = [s_1s_2 - v_1 \cdot v_2, s_1v_2 + s_2v_1 + v_1 \times v_2]
\]

\[
\frac{dq(t)}{dt} = \frac{1}{2}\omega(t)q(t) = \frac{1}{2} \begin{bmatrix} 0 & \omega(t) \end{bmatrix} q(t)
\]

• Back to rotation matrix
\[
R = \begin{pmatrix}
1 - 2v_y^2 - 2v_z^2 & 2v_xv_y - 2sv_z & 2v_xv_z + 2sv_y \\
2v_xv_y + 2sv_z & 1 - 2v_x^2 - 2v_z^2 & 2v_yv_z - 2sv_x \\
2v_xv_z - 2sv_y & 2v_yv_z + 2sv_x & 1 - 2v_x^2 - 2v_y^2
\end{pmatrix}
\]
Outline

• Rigid Body Representation
• Kinematics
• Dynamics
• Simulation Algorithm
• Collisions and Contact Response
How do $x(t)$ and $R(t)$ change over time?

Linear velocity $v(t)$ describes the velocity of the center of mass $x$ (m/s)

Angular velocity
Kinematics: Velocities

• Angular velocity, represented by $\omega(t)$
  – Direction: axis of rotation
  – Magnitude $|\omega|$: angular velocity about the axis (rad/s)

$$\dot{x} = \omega \times x$$

• Time derivative of rotation matrix:
  – Velocities of the body-frame axes, i.e. the columns of $R$

$$\dot{R} = \begin{pmatrix}
\omega(t) \times \begin{pmatrix}
r_{xx} \\
r_{xy} \\
r_{xz}
\end{pmatrix} & \omega(t) \times \begin{pmatrix}
r_{yx} \\
r_{yy} \\
r_{yz}
\end{pmatrix} & \omega(t) \times \begin{pmatrix}
r_{zx} \\
r_{zy} \\
r_{zz}
\end{pmatrix}
\end{pmatrix}$$
Angular Velocities

\[ \frac{d}{dt} \mathbf{R}(t) = \begin{pmatrix}
0 & -\omega_z(t) & \omega_y(t) \\
\omega_z(t) & 0 & -\omega_x(t) \\
-\omega_y(t) & \omega_x(t) & 0
\end{pmatrix} \mathbf{R}(t) = \omega(t)^* \mathbf{R}(t) \]
Outline

• Rigid Body Representation
• Kinematics
• Dynamics
• Simulation Algorithm
• Collisions and Contact Response
Dynamics: Accelerations

• How do $v(t)$ and $\omega(t)$ change over time?
• First we need some more machinery
  – Forces and Torques
  – Linear and angular momentum
  – Inertia Tensor
• Simplify equations by formulating accelerations in terms of momentum derivatives instead of velocity derivatives
Forces and Torques

- **External forces** $\mathbf{f}_i(t)$ act on particles
  - Total external force $\mathbf{F} = \sum \mathbf{f}_i(t)$
- **Torques** depend on distance from the center of mass:
  - Total external torque
    $$\tau(t) = \sum (((\mathbf{r}_i(t) - \mathbf{x}(t)) \times \mathbf{f}_i(t))$$
- $\mathbf{F}(t)$ doesn’t convey any information about where the various forces act
- $\tau(t)$ does tell us about the distribution of forces
Linear Momentum

- Linear momentum $P(t)$ lets us express the effect of total force $F(t)$ on body (due to conservation of energy):
  \[ F(t) = \frac{dP(t)}{dt} \]

- Linear momentum is the product of mass and linear velocity
  - $P(t) = \sum m_i \frac{dr_i(t)}{dt}$
    - $= \sum m_i v(t) + \omega(t) \times \sum m_i (r_i(t) - x(t))$
    - $= \sum m_i v(t) = M \mathbf{v}(t)$
  - Just as if body were a particle with mass $M$ and velocity $v(t)$
  - Time derivative of $\mathbf{v}(t)$ to express acceleration:
    \[ \dot{\mathbf{v}}(t) = M^{-1} \frac{dP(t)}{dt} = M^{-1} F(t) \]

- Use $P(t)$ instead of $v(t)$ in state vectors
Angular momentum

• Same thing, angular momentum \( L(t) \) allows us to express the effect of total torque \( \tau(t) \) on the body:

\[
\dot{L}(t) = \tau(t)
\]

• Similarly, there is a linear relationship between momentum and velocity:

\[
L(t) = I \omega(t)
\]

– \( I(t) \) is inertia tensor, plays the role of mass

• Use \( L(t) \) instead of \( \omega(t) \) in state vectors
Inertia Tensor

- 3x3 matrix describing how the shape and mass distribution of the body affects the relationship between the angular velocity and the angular momentum $L(t)$
- Analogous to mass – rotational mass
- We actually want the inverse $I^{-1}(t)$ to compute $\omega(t) = I^{-1}(t)L(t)$
Inertia Tensor

\[
I = \begin{pmatrix}
I_{xx} & -I_{xy} & -I_{xz} \\
-I_{yx} & I_{yy} & -I_{yz} \\
-I_{zx} & -I_{zy} & I_{zz}
\end{pmatrix}
\]

Bunch of volume integrals:

\[
I_{xx} = \int_V \rho(x, y, z) \left( y^2 + z^2 \right) dV \\
I_{yy} = \int_V \rho(x, y, z) \left( x^2 + z^2 \right) dV \\
I_{zz} = \int_V \rho(x, y, z) \left( x^2 + y^2 \right) dV \\
I_{xy} = I_{yx} = \int_V \rho(x, y, z) (xy) dV \\
I_{xz} = I_{zx} = \int_V \rho(x, y, z) (zx) dV \\
I_{yz} = I_{zy} = \int_V \rho(x, y, z) (yz) dV
\]
Inertia Tensor

- Avoid recomputing inverse of inertia tensor
- Compute $I$ in body space $I_{\text{body}}$ and then transform to world space as required
  - $I(t)$ varies in world space, but $I_{\text{body}}$ is constant in body space for the entire simulation
- Intuitively:
  - Transform $\omega(t)$ to body space, apply inertia tensor in body space, and transform back to world space
  - $L(t)=I(t)\omega(t)= R(t) \ I_{\text{body}} \ R^T(t) \ \omega(t)$
  - $I^{-1}(t)= R(t) \ I_{\text{body}}^{-1} \ R^T(t)$
Computing $I_{\text{body}}^{-1}$

- There exists an orientation in body space which causes $I_{xy}$, $I_{xz}$, $I_{yz}$ to all vanish
  - Diagonalize tensor matrix, define the eigenvectors to be the local body axes
  - Increases efficiency and trivial inverse
- Point sampling within the bounding box
- Projection and evaluation of Greene’s thm.
  - Code implementing this method exists
  - Refer to Mirtich’s paper at http://www.acm.org/jgt/papers/Mirtich96
Approximation w/ Point

- **Pros:** Simple, fairly accurate, no B-rep needed.
- **Cons:** Expensive, requires volume test.
Use of Green’s Theorem

- **Pros**: Simple, exact, no volumes needed.
- **Cons**: Requires boundary representation.
Outline

- Rigid Body Representation
- Kinematics
- Dynamics
- Simulation Algorithm
- Collisions and Contact Response
Position state vector

\[ \dot{X}(t) = \frac{d}{dt} \begin{pmatrix} x(t) \\ q(t) \\ P(t) \\ L(t) \end{pmatrix} \]

\[ \begin{array}{c} \text{Spatial information} \\ \text{Velocity information} \end{array} \]

\( v(t) \) replaced by linear momentum \( P(t) \)

\( \omega(t) \) replaced by angular momentum \( L(t) \)

Size of the vector: \((3+4+3+3)N = 13N\)
Velocity state vector

\[ \dot{X}(t) = \frac{d}{dt} \begin{pmatrix} x(t) \\ q(t) \\ P(t) \\ L(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ \frac{1}{2} \omega(t)q(t) \\ F(t) \\ \tau(t) \end{pmatrix} = \begin{pmatrix} \frac{P(t)}{M} \\ \frac{1}{2} I^{-1} L(t)q(t) \\ F(t) \\ \tau(t) \end{pmatrix} \]

Conservation of momentum \((P(t), L(t))\) lets us express the accelerations in terms of forces and torques.
Simulation Algorithm

Pre-compute:
\[ M \leftarrow \sum m_i \]
\[ I_{\text{body}} \]

Initialize
\[ x, v, R, \omega, X, \dot{X} \]
\[ I^{-1} \leftarrow RI_{\text{body}}R^T \]
\[ L \leftarrow I\omega \]

Accumulate forces
\[ \tau \leftarrow \sum r_i \times f_i \]
\[ F \leftarrow \sum f_i \]

(\(X, \dot{X}\) \leftarrow \text{step}(X, \dot{X}, F, \tau)\)
\[ R \leftarrow \text{quat2mat}(q) \]
\[ I^{-1} \leftarrow RI_{\text{body}}R^T \]

Your favorite ODE solver
Simulation Algorithm

Pre-compute:
\[ M \leftarrow \sum m_i \]
\[ I_{\text{body}} \]

Initialize
\[ x, v, R, \omega \]
\[ I^{-1} \leftarrow RI_{\text{body}}R^T \]
\[ L \leftarrow I\omega \]

Accumulate forces
\[ \tau \leftarrow \sum r_i \times f_i \]
\[ F \leftarrow \sum f_i \]
\[ P \leftarrow P + \Delta tF \]
\[ L \leftarrow L + \Delta t\tau \]
\[ \omega \leftarrow I^{-1}L \]
\[ x \leftarrow x + \Delta t \frac{P}{M} \]
\[ q \leftarrow q + \Delta t \frac{1}{2} \omega q \]
\[ R \leftarrow \text{quat2mat}(q) \]
\[ I^{-1} \leftarrow RI_{\text{body}}R^T \]

Explicit Euler step
Outline

- Rigid Body Representation
- Kinematics
- Dynamics
- Simulation Algorithm
- Collision Detection and Contact Determination
  - Contact classification
  - Intersection testing, bisection, and nearest features
What happens when bodies collide?

- **Colliding**
  - Bodies bounce off each other
  - Elasticity governs ‘bounciness’
  - Motion of bodies changes *discontinuously* within a discrete time step
  - ‘Before’ and ‘After’ states need to be computed

- **In contact**
  - Resting
  - Sliding
  - Friction
Detecting collisions and response

• Several choices
  – Collision detection: which algorithm?
  – Response: Backtrack or allow penetration?

• Two primitives to find out if response is necessary:
  – Distance(A,B): cheap, no contact information → fast intersection query
  – Contact(A,B): expensive, with contact information
Distance(A,B)

- Returns a value which is the minimum distance between two bodies
- Approximate may be ok
- Negative if the bodies intersect
- Convex polyhedra
  - Lin-Canny and GJK -- 2 classes of algorithms
- Non-convex polyhedra
  - Much more useful but hard to get distance fast
  - PQP/RAPID/SWIFT++
- Remark: most of these algorithms give inaccurate information if bodies intersect, except for DEEP
Contacts(A,B)

- Returns the set of features that are nearest for disjoint bodies or intersecting for penetrating bodies
- Convex polyhedra
  - LC & GJK give the nearest features as a bi-product of their computation – only a single pair. Others that are equally distant may not be returned.
- Non-convex polyhedra
  - Much more useful but much harder problem especially contact determination for disjoint bodies
  - Convex decomposition: SWIFT++
Prereq: Fast intersection test

• First, we want to make sure that bodies will intersect at next discrete time instant

• If not:
  – $X_{\text{new}}$ is a valid, non-penetrating state, proceed to next time step

• If intersection:
  – Classify contact
  – Compute response
  – Recompute new state
Bodies intersect → classify contacts

- **Colliding contact (‘easy’)**
  - $v_{\text{rel}} < -\varepsilon$
  - Instantaneous change in velocity
  - Discontinuity: requires restart of the equation solver

- **Resting contact (hard!)**
  - $-\varepsilon < v_{\text{rel}} < \varepsilon$
  - Gradual contact forces avoid interpenetration
  - No discontinuities

- **Bodies separating**
  - $v_{\text{rel}} > \varepsilon$
  - No response required
Colliding contacts

- At time $t_i$, body A and B intersect and $v_{\text{rel}} < -\varepsilon$
- Discontinuity in velocity: need to stop numerical solver
- Find time of collision $t_c$
- Compute new velocities $v^+(t_c) \rightarrow X^+(t)$
- Restart ODE solver at time $t_c$ with new state $X^+(t)$
Time of collision

- We wish to compute when two bodies are “close enough” and then apply contact forces
- Let’s recall a particle colliding with a plane
Time of collision

- We wish to compute $t_c$ to some tolerance
Time of collision

1. A common method is to use **bisection search** until the distance is positive but less than the tolerance

2. Use **continuous collision detection**

3. $t_c$ not always needed
   \[ \rightarrow \text{penalty-based methods} \]
Bisection

findCollisionTime(X,t,\Delta t)
    foreach pair of bodies (A,B) do
        Compute_New_Body_States(S_{copy}, t, \Delta t);
        hs(A,B) = \Delta t;  // H is the target timestep
        if Distance(A,B) < 0 then
            try_h = \Delta t /2;  try_t = t + try_h;
            while TRUE do
                Compute_New_Body_States(S_{copy}, t, try_t - t);
                if Distance(A,B) < 0 then
                    try_h /= 2;  try_t -= try_h;
                else if Distance(A,B) < \varepsilon then
                    break;
                else
                    try_h /= 2;  try_t += try_h;
                hs(A,B)->append(try_t – t);
            h = min( hs );
What happens upon collision

- **Force driven**
  - Penalty based
  - Easier, but slow objects react ‘slow’ to collision

- **Impulse driven**
  - Impulses provide instantaneous changes to velocity, unlike forces
    \[ \Delta(P) = J \]
  - We apply impulses to the colliding objects, at the point of collision
  - For frictionless bodies, the direction will be the same as the normal direction:
    \[ J = j \cdot n \]
Colliding Contact Response

• Assumptions:
  – Convex bodies
  – Non-penetrating
  – Non-degenerate configuration
    • edge-edge or vertex-face
    • appropriate set of rules can handle the others

• Need a contact unit normal vector
  – Face-vertex case: use the normal of the face
  – Edge-edge case: use the cross-product of the direction vectors of the two edges
Colliding Contact Response

- Point velocities at the nearest points:
  \[ \dot{p}_a(t_0) = v_a(t_0) + \omega_a(t_0) \times (p_a(t_0) - x_a(t_0)) \]
  \[ \dot{p}_b(t_0) = v_b(t_0) + \omega_b(t_0) \times (p_b(t_0) - x_b(t_0)) \]

- Relative contact normal velocity:
  \[ v_{rel} = \hat{n}(t_0) \cdot (\dot{p}_a(t_0) - \dot{p}_b(t_0)) \]
Colliding Contact Response

• We will use the empirical law of frictionless collisions: $v_{rel}^+ = -\epsilon v_{rel}^-$

  – Coefficient of restitution $\epsilon \in [0,1]$
    • $\epsilon = 0$ – bodies stick together
    • $\epsilon = 1$ – loss-less rebound

• After some manipulation of equations...

$$ j = \frac{-(1 + \epsilon)v_{rel}^-}{\frac{1}{M_a} + \frac{1}{M_b} + \hat{n}(t_0) \cdot (I_a^{-1}(t_0) (r_a \times \hat{n}(t_0))) \times r_a + \hat{n}(t_0) \cdot (I_b^{-1}(t_0) (r_b \times \hat{n}(t_0))) \times r_b} $$
Compute and apply impulses

- The impulse is an instantaneous force – it changes the velocities of the bodies instantaneously:

\[ J = jn \]

\[ \Delta v = \frac{J}{M} \]

\[ \Delta L = (x_{\text{impact}} - x) \times J \]
Penalty Methods

• If we don’t look for time of collision $t_c$ then we have a simulation based on penalty methods: the objects are allowed to intersect.

• **Global** or **local** response
  - **Global**: The penetration depth is used to compute a spring constant which forces them apart (dynamic springs)
  - **Local**: Impulse-based techniques
References

- COMP259 Rigid Body Simulation Slides, Chris Vanderknyff 2004
- Rigid Body Dynamics (course slides), M Müller-Fischer 2005, ETHZ Zurich