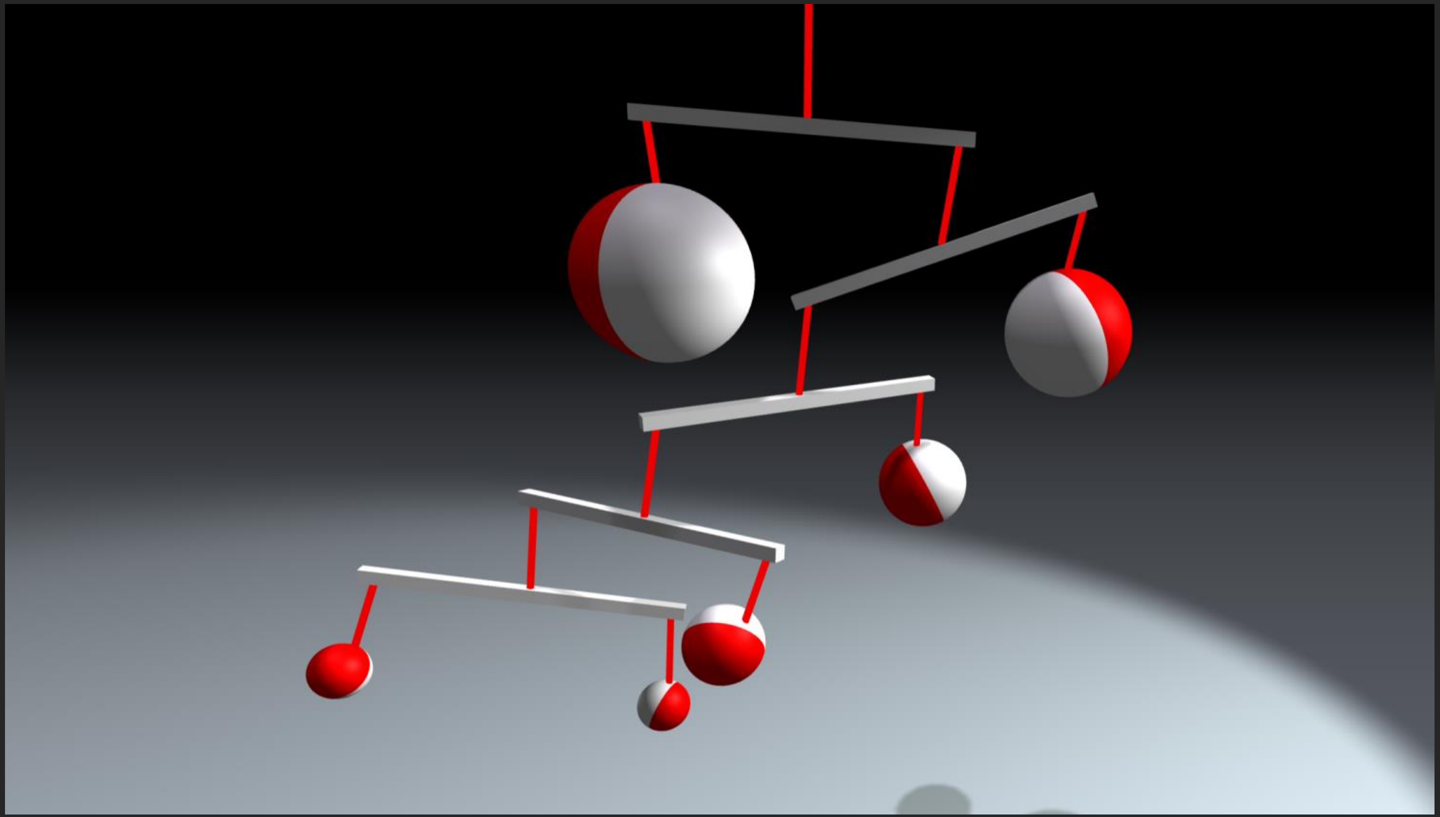
Basic Rigid Body Simulation



Matthias Müller, Ten Minute Physics

matthiasmueller.info/tenMinutePhysics

PhysX
by NVIDIA

NVIDIA /warp

C++

Method

To simulate rigid bodies…

solve

$$0 \le \begin{bmatrix} {}^{u}\mathbf{J}_n^{(\ell)}\mathbf{u}^{(\ell+1)} + \frac{{}^{u}\mathbf{C}_n^{(\ell)}}{\Delta t} + \frac{\partial\, {}^{u}\mathbf{C}_n^{(\ell)}}{\partial t} \\ {}^{u}\mathbf{D}^T\, {}^{u}\mathbf{J}_f \mathbf{u}^{(\ell+1)} + {}^{u}\mathbf{E}\, {}^{u}\beta \\ {}^{b}\mathbf{D}^T\, {}^{b}\mathbf{J}_f \mathbf{u}^{(\ell+1)} + {}^{b}\mathbf{E}\, {}^{b}\beta \\ \mathbf{U}\, {}^{u}\mathbf{p}_n^{(\ell+1)} - {}^{u}\mathbf{E}^T\, {}^{u}\alpha \\ {}^{b}\mathbf{p}_{f\max} - {}^{b}\mathbf{E}^T\, {}^{b}\alpha \end{bmatrix} \perp \begin{bmatrix} {}^{u}\mathbf{p}^{(\ell+1)} \\ {}^{u}\alpha \\ {}^{b}\alpha \\ {}^{u}\beta \\ {}^{b}\beta \end{bmatrix} \ge 0.$$

,where

$$\widehat{{}^{\kappa}C_{i\sigma}}(\tilde{\mathbf{q}},\tilde{t}) = {}^{\kappa}C_{i\sigma}(\mathbf{q},t) + \frac{\partial\, {}^{\kappa}C_{i\sigma}}{\partial \mathbf{q}}\Delta\mathbf{q} + \frac{\partial\, {}^{\kappa}C_{i\sigma}}{\partial t}\Delta t + \frac{1}{2}\left((\Delta\mathbf{q})^T \frac{\partial^2\, {}^{\kappa}C_{i\sigma}}{\partial \mathbf{q}^2}\Delta\mathbf{q} + 2\frac{\partial^2\, {}^{\kappa}C_{i\sigma}}{\partial \mathbf{q}\partial t}\Delta\mathbf{q}\Delta t + \frac{\partial^2\, {}^{\kappa}C_{i\sigma}}{\partial t^2}\Delta t^2\right)$$

$${}^{\kappa}\mathbf{J}_{i\sigma} = \frac{\partial({}^{\kappa}\mathbf{C}_{i\sigma})}{\partial \mathbf{q}}\mathbf{H}$$

$${}^{\kappa}\mathbf{k}_{i\sigma}(\mathbf{q},\mathbf{u},t) = \frac{\partial({}^{\kappa}\mathbf{C}_{i\sigma})}{\partial \mathbf{q}}\frac{\partial \mathbf{H}}{\partial t}\mathbf{u} + \frac{\partial^2({}^{\kappa}\mathbf{C}_{i\sigma})}{\partial \mathbf{q}\partial t}\mathbf{H}\mathbf{u} + \frac{\partial^2({}^{\kappa}\mathbf{C}_{i\sigma})}{\partial t^2},$$

Is rigid body simulation only for math wizards?

Using Position Based Dynamics

See tutorials:

- Position Based Dynamics (9)
- 3d Vector Math (7)

PBD Algorithm for Particles

while simulating
    for all particles i
        $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t\ \mathbf{g}$
        $\mathbf{p}_i \leftarrow \mathbf{x}_i$
        $\mathbf{x}_i \leftarrow \mathbf{x}_i + \Delta t\ \mathbf{v}_i$

    for all constraints C
        solve($C, \Delta t$)

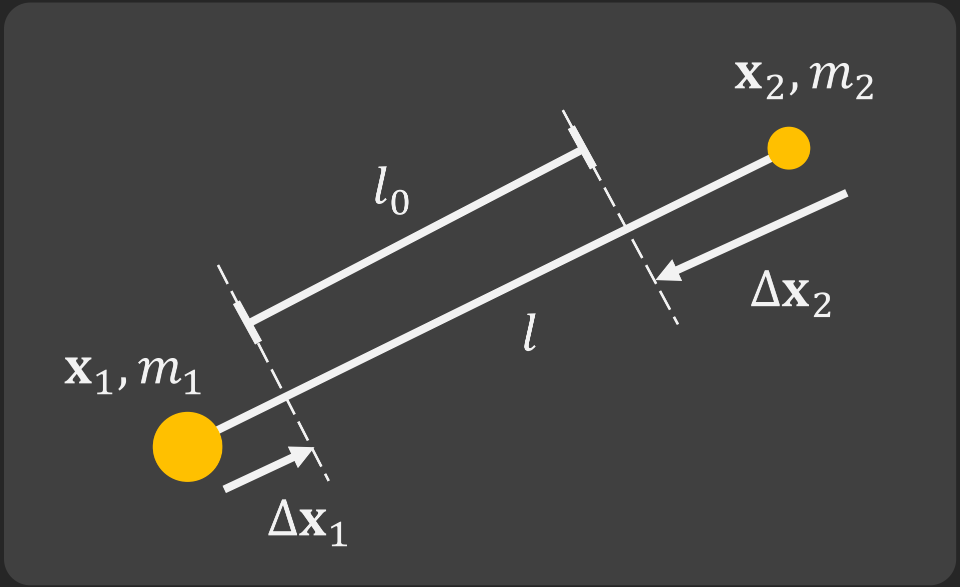
    for all particles i
        $\mathbf{v}_i \leftarrow (\mathbf{x}_i - \mathbf{p}_i)/\Delta t$

solve($C, \Delta t$):

for all particles i in C
    compute $\Delta\mathbf{x}_i$
    $\mathbf{x}_i \leftarrow \mathbf{x}_i + \Delta\mathbf{x}_i$

Distance Constraint

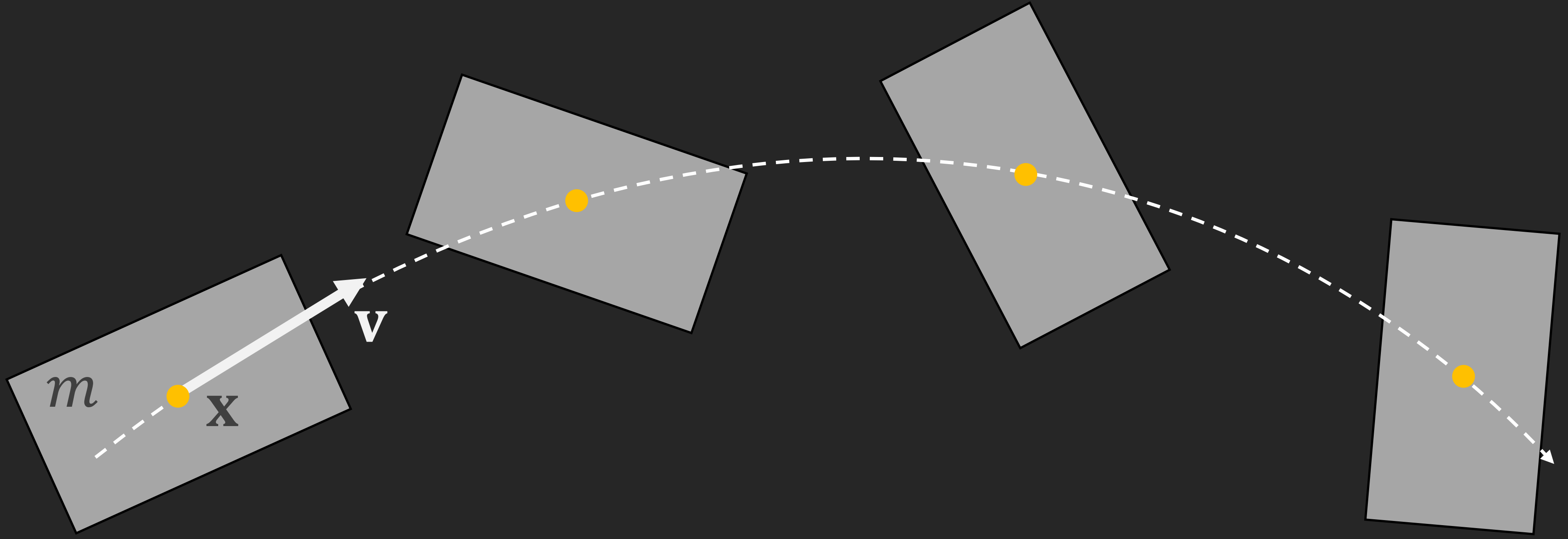
- Rest distance l_0
- Current distance l
- Masses m_i
- Inverse masses $w_i = 1/m_1$



$$\Delta\mathbf{x}_1 = \frac{w_1}{w_1 + w_2}(l - l_0)\frac{\mathbf{x}_2 - \mathbf{x}_1}{|\mathbf{x}_2 - \mathbf{x}_1|}$$

$$\Delta\mathbf{x}_2 = -\frac{w_2}{w_1 + w_2}(l - l_0)\frac{\mathbf{x}_2 - \mathbf{x}_1}{|\mathbf{x}_2 - \mathbf{x}_1|}$$

Rigid Bodies

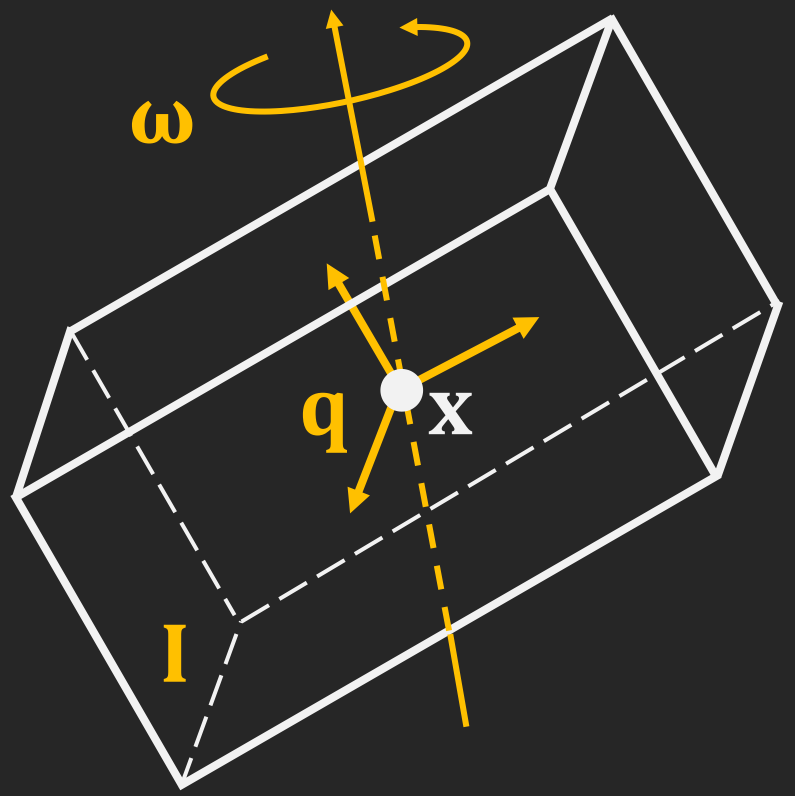


- The center of mass of a rigid body acts like a particle with mass m, position $\mathbf{x}$ and velocity $\mathbf{v}$.

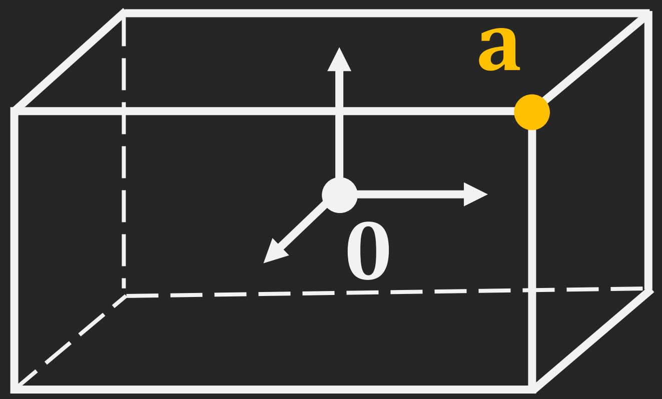
Orientational Quantities

A rigid body also has

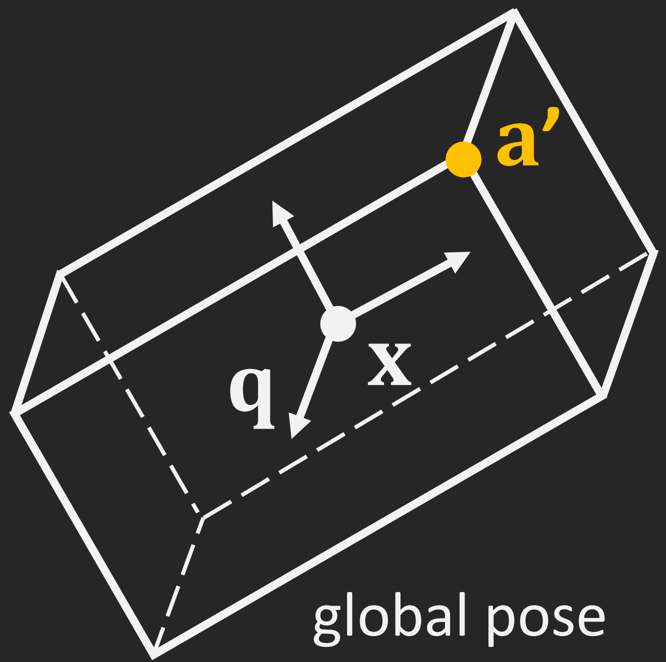
- an orientation $\mathbf{q}$
- an angular velocity $\boldsymbol{\omega}$
- and the moment of inertia $\mathbf{I}$



3d Rigid Transformation



local frame (center at the origin)



global pose

$$\mathbf{a}' = \mathbf{x} + \mathbf{q} * \mathbf{a}$$

$$\mathbf{a} = \mathbf{q}^{-1} * (\mathbf{a}' - \mathbf{x})$$

$\mathbf{q}$ is a quaternion, $*$ is quaternion rotation

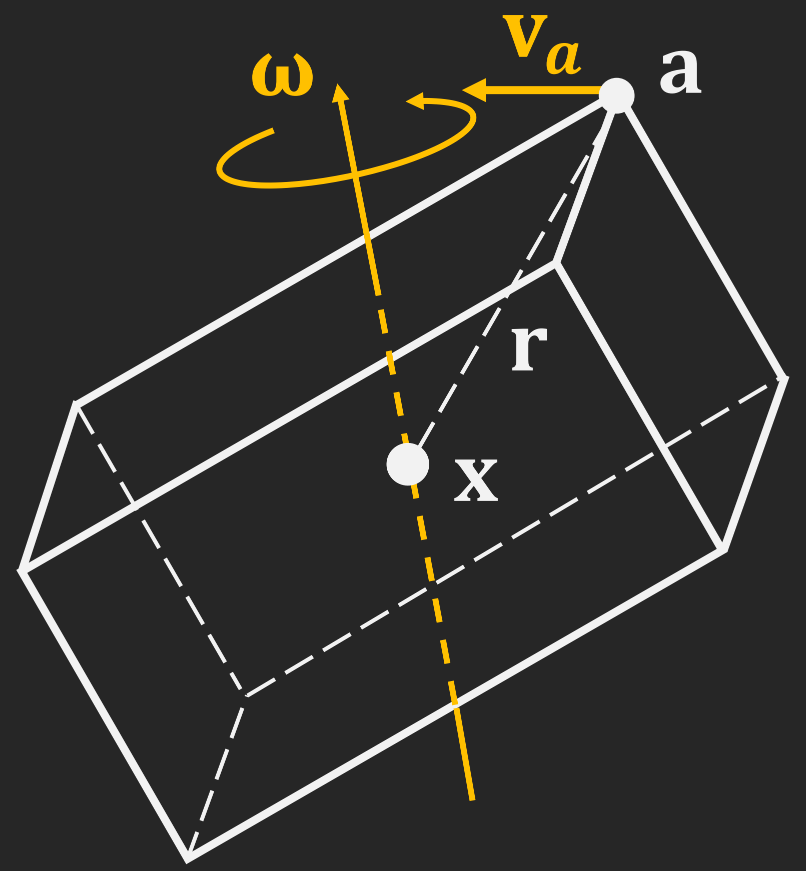
Three.js

```
this.rot = new THREE.Quaternion();
this.rot.setFromEuler(new THREE.Euler(angles.x, angles.y, angles.z));
this.invRot = this.rot.clone();
this.invRot.invert();
```

```
localToWorld(localPos, worldPos)
{
    worldPos.copy(localPos);
    worldPos.applyQuaternion(this.rot);
    worldPos.add(this.pos);
}

worldToLocal(worldPos, localPos)
{
    localPos.copy(worldPos);
    localPos.sub(this.pos);
    localPos.applyQuaternion(this.invRot);
}
```

Angular Velocity

- $\boldsymbol{\omega}$ is a 3d vector passing through $\mathbf{x}$
- Its length $|\boldsymbol{\omega}|$ is the speed in angle per second
- Its direction describes the axis of rotation
- The velocity of a point $\mathbf{a}$ is $\mathbf{v}_a = \boldsymbol{\omega} \times \mathbf{r}$
- With moving body: $\mathbf{v}_a = \mathbf{v} + \boldsymbol{\omega} \times \mathbf{r}$



Moment of Inertia

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

$$\mathbf{a} = 1/m \cdot \mathbf{f}$$

- force causes acceleration
- mass is the resistance to force

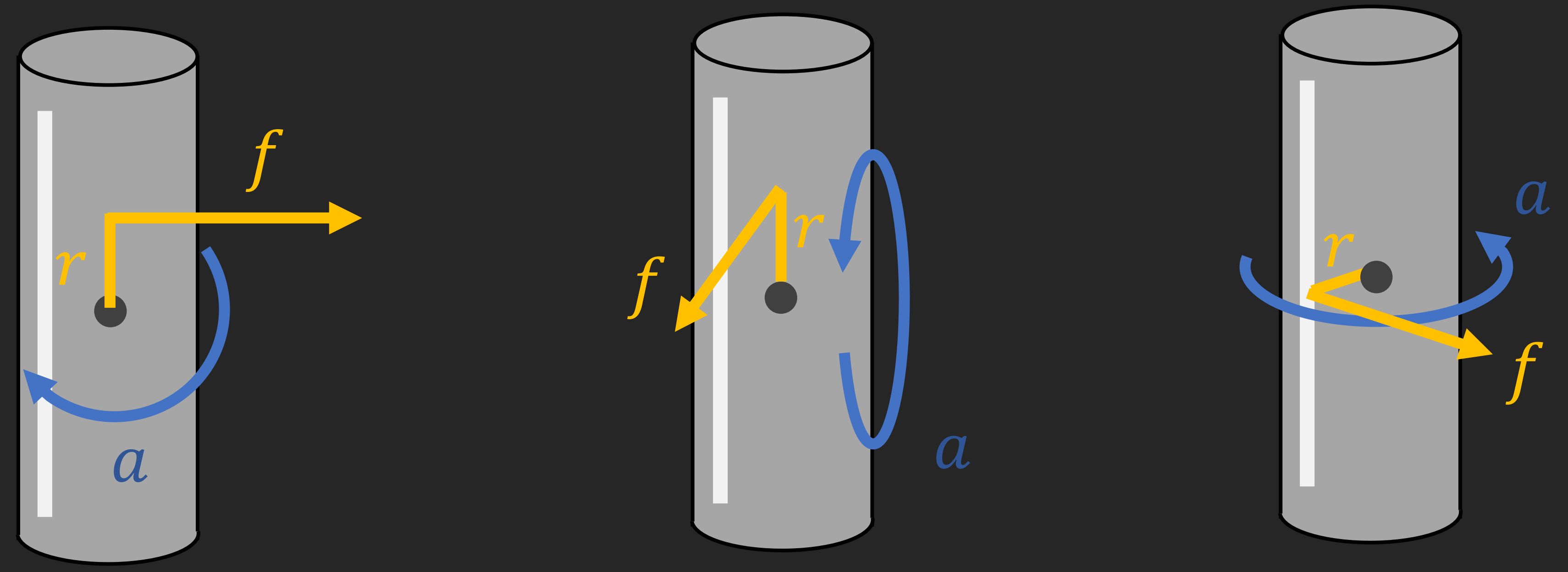
$$\boldsymbol{\tau} = \mathbf{I} \cdot \boldsymbol{\alpha}$$

$$\boldsymbol{\alpha} = \mathbf{I}^{-1} \cdot \boldsymbol{\tau}$$

- torque (angular force $\mathbf{r} \times \mathbf{f}$) causes angular acceleration
- Moment of inertia describes the resistance to torque

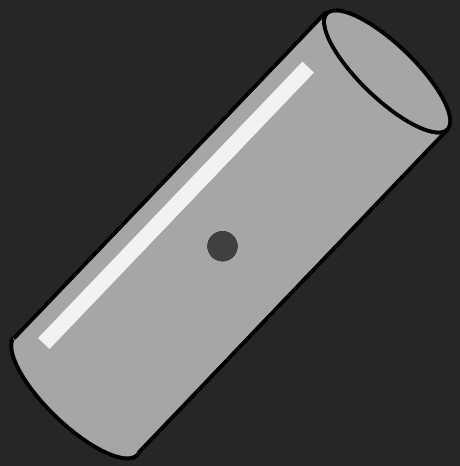
Moment of Inertia

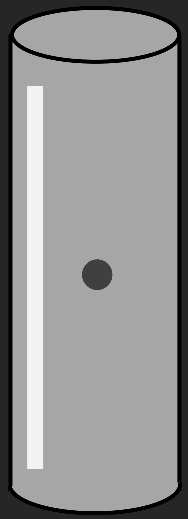
- The resistance to a torque of the same object can vary in different directions:



large resistance | large resistance | small resistance

The Inertia Tensor





$$\boldsymbol{\tau} = \mathbf{I} \cdot \boldsymbol{\alpha}$$

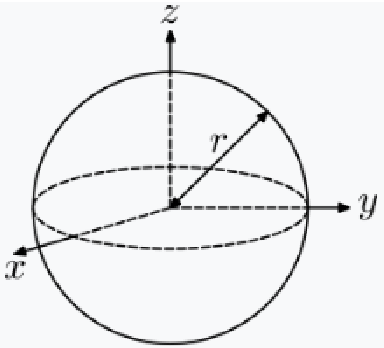
$$\mathbf{I} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{xz} & I_{yz} & I_{zz} \end{bmatrix}$$

$$\mathbf{I} = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix}$$

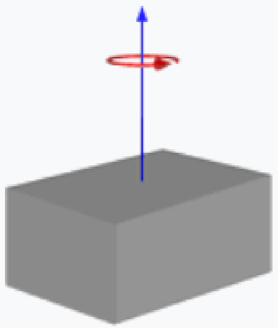
aligned with principal axis

Wikipedia

- For basic shapes see https://en.wikipedia.org/wiki/List_of_moments_of_inertia



$$I = \frac{2}{3} m r^2$$



$$I_h = \frac{1}{12} m \left(w^2 + d^2\right)$$

$$I_w = \frac{1}{12} m \left(d^2 + h^2\right)$$

$$I_d = \frac{1}{12} m \left(w^2 + h^2\right)$$

- For general triangle meshes, see an upcoming tutorial

PBD Algorithm for Rigid Bodies

while simulating
    for all bodies i
        integrate $\mathbf{v}_i, \mathbf{x}_i$
        integrate $\boldsymbol{\omega}_i, \mathbf{q}_i$

    for all constraints C
        solve$(C, \Delta t)$

    for all bodies i
        update $\mathbf{v}_i$
        update $\boldsymbol{\omega}_i$

solve$(C, \Delta t)$:

for all bodies i in C
    compute $\Delta\mathbf{x}_i, \Delta\mathbf{q}_i$
    $\mathbf{x}_i \leftarrow \mathbf{x}_i + \Delta\mathbf{x}_i$
    $\mathbf{q}_i \leftarrow \mathbf{q}_i + \Delta\mathbf{q}_i$

PBD Integration

for all bodies i

$$\mathbf{p}_i \leftarrow \mathbf{x}_i$$
$$\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t\ \mathbf{g}$$
$$\mathbf{x}_i \leftarrow \mathbf{x}_i + \Delta t\ \mathbf{v}_i$$
$$\mathbf{q}_{\text{prev}} \leftarrow \mathbf{q}$$
$$\boldsymbol{\omega} \leftarrow \boldsymbol{\omega} + h\mathbf{I}^{-1}\boldsymbol{\tau}_{\text{ext}}$$
$$\mathbf{q} \leftarrow \mathbf{q} + ½h\mathbf{v}[\omega_x, \omega_x, \omega_x, 0]\mathbf{q}$$

```
integrate(dt, gravity)
{
    // linear motion
    this.prevPos.copy(this.pos);
    this.vel.addScaledVector(gravity, dt);
    this.pos.addScaledVector(this.vel, dt);

    // angular motion
    this.prevRot.copy(this.rot);
    this.dRot.set(
        this.omega.x,
        this.omega.y,
        this.omega.z,
        0.0
    );
    this.dRot.multiply(this.rot);
    this.rot.x += 0.5 * dt * this.dRot.x;
    this.rot.y += 0.5 * dt * this.dRot.y;
    this.rot.z += 0.5 * dt * this.dRot.z;
    this.rot.w += 0.5 * dt * this.dRot.w;
    this.rot.normalize();
}
```

PBD Velocity Update

for all bodies i

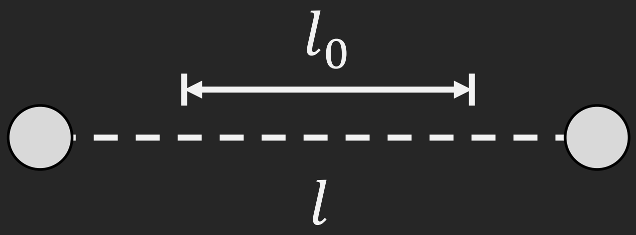
$$\mathbf{v}_i \leftarrow (\mathbf{x}_i - \mathbf{p}_i)/\Delta t$$

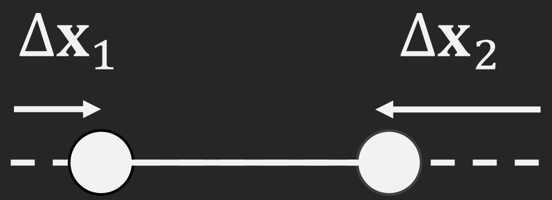
$$\Delta\mathbf{q} \leftarrow \mathbf{q}\mathbf{q}_{\text{prev}}^{-1}$$

$$\boldsymbol{\omega} \leftarrow 2[\Delta q_x, \Delta q_x, \Delta q_x]/\Delta t$$

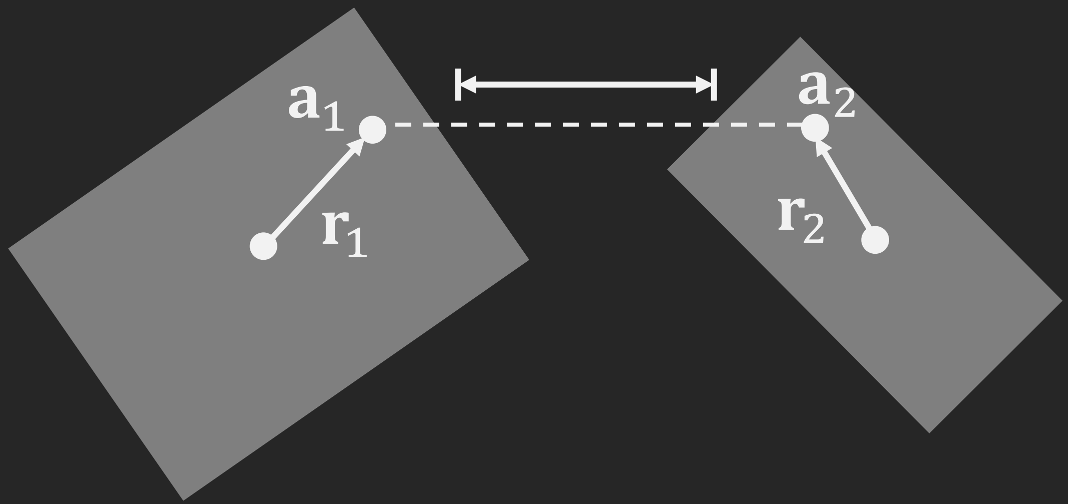
```
updateVelocities()
{
    // linear motion
    this.vel.subVectors(this.pos, this.prevPos);
    this.vel.multiplyScalar(1.0 / this.dt);

    // angular motion
    this.prevRot.invert();
    this.dRot.multiplyQuaternions(this.rot, this.prevRot);
    this.omega.set(
        this.dRot.x * 2.0 / this.dt,
        this.dRot.y * 2.0 / this.dt,
        this.dRot.z * 2.0 / this.dt
    );
    if (this.dRot.w < 0.0)
        this.omega.negate();
}
```

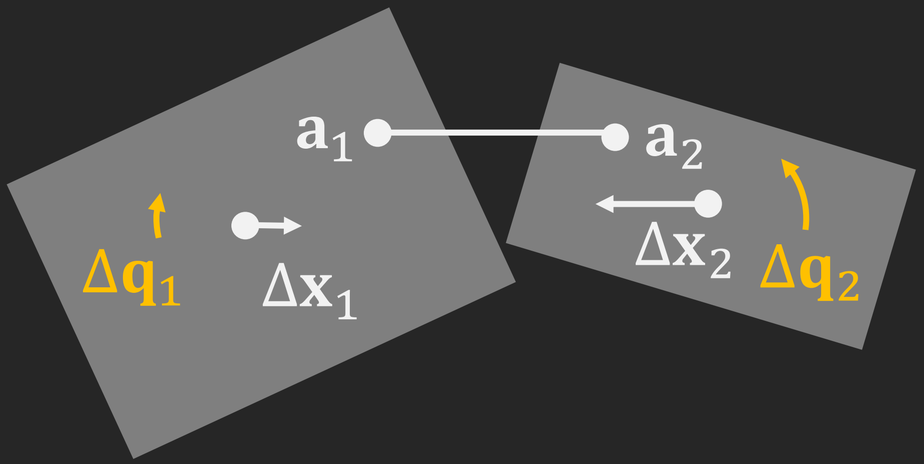
Distance Constraint





corrections proportional to m^{-1}





corrections proportional to $m^{-1}$and $\mathbf{I}^{-1}$

XPBD Algorithm for Rigid Bodies

- Given $\mathbf{r}_1, \mathbf{r}_2$, constraint direction $\mathbf{n}$ and the constraint distance C
- For a distance constraint: $\mathbf{n} = (\mathbf{a}_2 - \mathbf{a}_1)/|\mathbf{a}_2 - \mathbf{a}_1|$ and $C = l - l_0$
- Compute generalized inverse masses:

$$w_i \leftarrow m_i^{-1} + (\mathbf{r}_i \times \mathbf{n})^{\mathrm{T}} \mathbf{I}_i^{-1} (\mathbf{r}_i \times \mathbf{n})$$

- Compute Lagrange multiplier (α physical inverse stiffness):

$$\lambda \leftarrow -C \cdot (w_1 + w_2 + \frac{\alpha}{\Delta t^2})^{-1}$$

- Update states:

$$\mathbf{x}_i \leftarrow \mathbf{x}_i \pm w_i \lambda \mathbf{n}$$

$$\mathbf{q}_i \leftarrow \mathbf{q}_i \pm \frac{1}{2} \lambda [\mathbf{I}_i^{-1} (\mathbf{r}_i \times \mathbf{n}), 0]\, \mathbf{q}_i$$

$\lambda \mathbf{n}/\Delta t^2$ is the constraint force

PBD vs. XPBD

Both are unconditionally stable (never blow up)

PBD: simply scaling the correction vector

$$\lambda \leftarrow -s \cdot C \cdot (w_1 + w_2)^{-1}$$

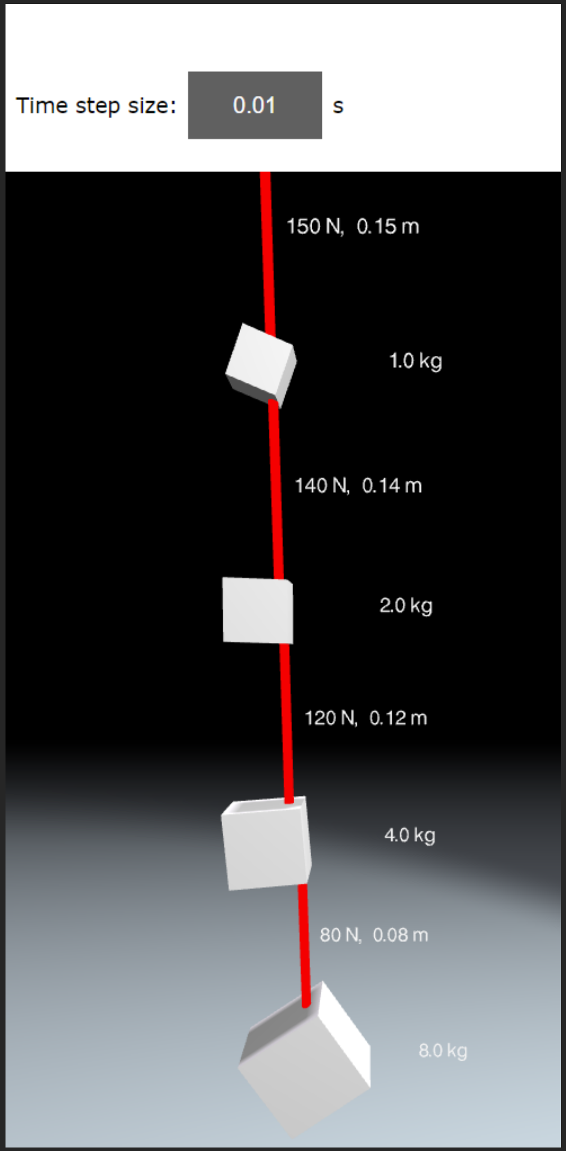
- Time-step dependent
- Scaling factor s is a non-physical quantity

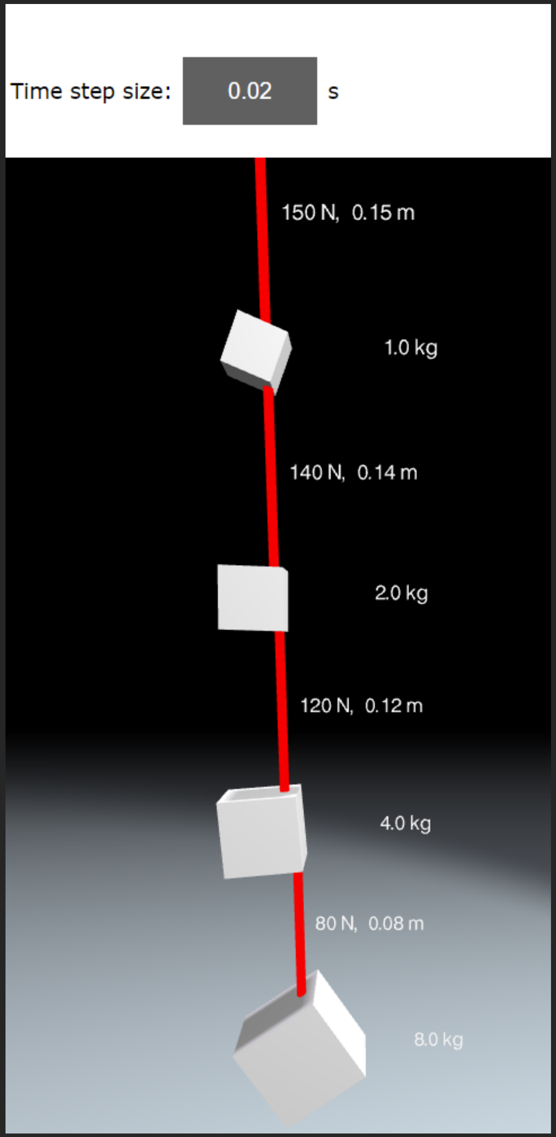
XPBD: derived from implicit Euler integration

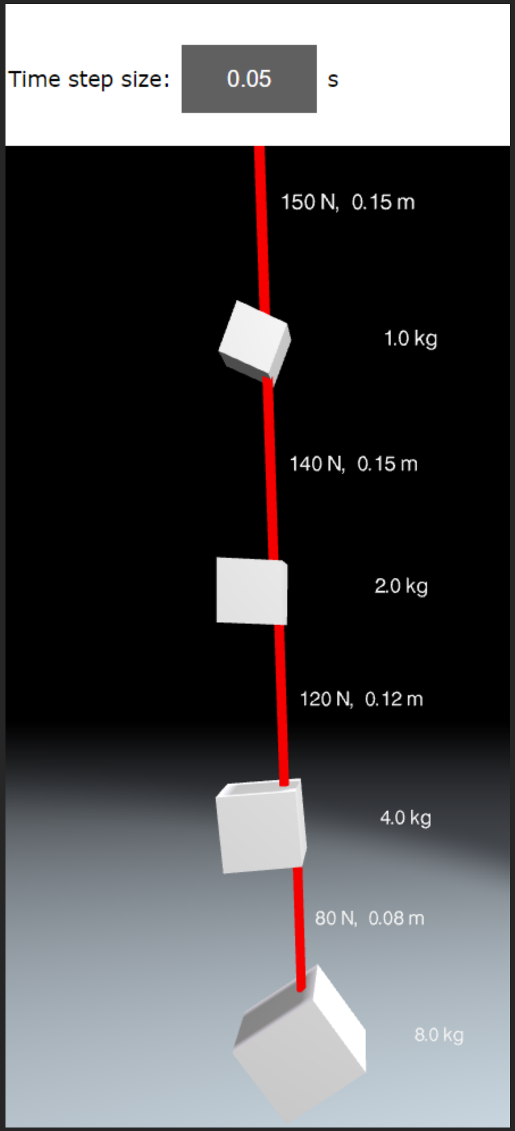
$$\lambda \leftarrow -C \cdot (w_1 + w_2 + \frac{\alpha}{\Delta t^2})^{-1}$$

- Time step independent
- The scalar α is the inverse of physical stiffness
- Both can handle infinite stiffness with $s = 1$ and $\alpha = 0$!
- For infinite stiffness they are identical

Chain Demo







$$g = 10.0\ \frac{m}{s^2}$$

Three.js

```
applyCorrection(compliance, corr, pos, otherBody, otherPos)
{
    if (corr.lengthSq() == 0.0)
        return;

    let C = corr.length();
    let normal = corr.clone();
    normal.normalize();

    let w = this.getInverseMass(normal, pos);
    if (otherBody != undefined)
        w += otherBody.getInverseMass(normal, otherPos);

    if (w == 0.0)
        return;

    // XPBD
    let alpha = compliance / this.dt / this.dt;
    let lambda = -C / (w + alpha);
    normal.multiplyScalar(-lambda);

    this._applyCorrection(normal, pos);
    if (otherBody != undefined) {
        normal.multiplyScalar(-1.0);
        otherBody._applyCorrection(normal, otherPos);
    }
}
```

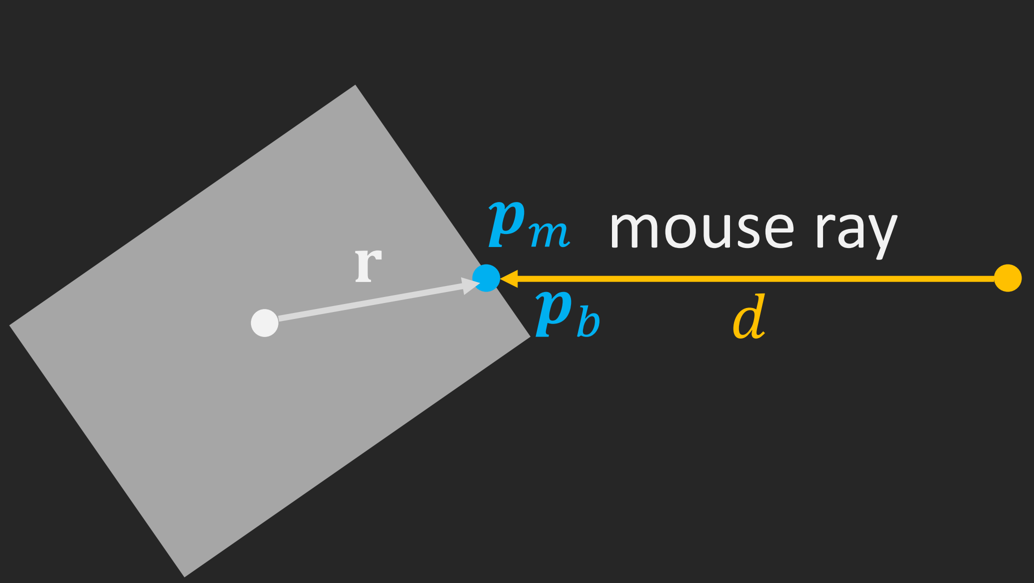
Three.js

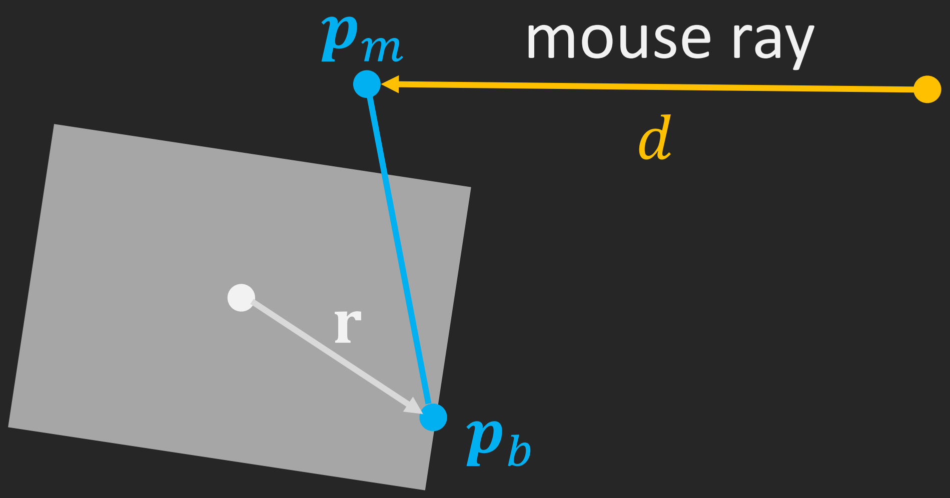
```
getInverseMass(normal, pos)
{
    if (this.invMass == 0.0)
        return 0.0;

    let rn = normal.clone();

    rn.subVectors(pos, this.pos);
    rn.cross(normal);
    rn.applyQuaternion(this.invRot);

    let w =
        rn.x * rn.x * this.invInertia.x +
        rn.y * rn.y * this.invInertia.y +
        rn.z * rn.z * this.invInertia.z;

    w += this.invMass;

    return w;
}
```

Three.js

```
_applyCorrection(corr, pos)
{
    if (this.invMass == 0.0)
        return;

    // linear correction

    this.pos.addScaledVector(corr, this.invMass);

    // angular correction

    let dOmega = corr.clone();
    dOmega.subVectors(pos, this.pos);
    dOmega.cross(corr);
    dOmega.applyQuaternion(this.invRot);
    dOmega.multiply(this.invInertia);
    dOmega.applyQuaternion(this.rot);

    this.dRot.set(dOmega.x, dOmega.y, dOmega.z, 0.0);
    this.dRot.multiply(this.rot);
    this.rot.x += 0.5 * this.dRot.x;
    this.rot.y += 0.5 * this.dRot.y;
    this.rot.z += 0.5 * this.dRot.z;
    this.rot.w += 0.5 * this.dRot.w;
    this.rot.normalize();
    this.invRot.copy(this.rot);
    this.invRot.invert();
}
```

Interaction





On mouse down

- Intersect mouse ray with the scene to find $\mathbf{p}$
- Store the distance d along the ray
- Store the local position $\mathbf{r}$ on the body
- Create a distance constraint

On mouse move

- Update $\boldsymbol{p}_m$ using d
- Update $\boldsymbol{p}_b$ using $\mathbf{r}$ and the current pose of the body

See you in the next tutorial…