The motion of a rigid body is divided into two parts: linear motion, which is the motion of the mass center, and rotational motion.

1 Linear Motion

In its local space, a rigid body is made of a list of vertices: \{p_0, p_1, \ldots, p_n\}. Let \(m_i\) be the mass of vertex \(i\).

**Total mass.**

\[
M = \sum m_i.
\]  

(1)

**Center of mass.**

\[
x(t) = \sum m_i p_i(t).
\]  

(2)

In the local space, we have \(\sum m_i p_i = 0\). If not, the mesh needs to be centered at the origin first.

**Linear velocity.**

\[
v(t) = \dot{x}(t).
\]  

(3)

**Linear acceleration.** According to Newton’s law, we have:

\[
a(t) = v(t) = \frac{1}{M} f(t).
\]  

(4)

in which \(f\) is total force applied on the rigid body.

**Force.** The total force \(f\) is the sum of various forces. If we consider the gravity force and the damping force only, we have:

\[
f(t) = f_{gravity}(t) + f_{damping}(t) = Mg - kv(t).
\]  

(5)

in which \(g\) is the gravity acceleration and \(k\) is a damping constant.

**Velocity and position update.** Given \(x(t)\) and \(v(t)\), we first update \(v(t)\) to \(v(t + \Delta t)\) by forward Euler method:

\[
v(t + \Delta t) = v(t) + \frac{1}{M} \int_{t}^{t+\Delta t} f(s) ds = v(t) + \frac{\Delta t}{M} f(t).
\]  

(6)

We then update the position by:

\[
x(t + \Delta t) = x(t) + \int_{t}^{t+\Delta t} v(s) ds = x(t) + \Delta t v(t + \Delta t).
\]  

(7)

**Summary.** The algorithm works in three steps: first calculate the force, then update the velocity, and finally update the position. There is no need to define \(v(t)\) and \(v(t + \Delta t)\) as two separate vectors. We can just define a single vector \(v\) as the velocity and update it. Similarly, we can define \(x\) as the position vector and update it.

2 Rotational Motion

For rotation motion, we need to define the rotational “position”, “velocity”, and “force”. The “rotational force” is called torque, and the “rotational position” is called angular velocity. The question is what is the “rotation position”, or in other words, **how do we represent the orientation of a rigid body?** We could use Euler angles. But it is difficult to build the relationship with the angular velocity! We could use a matrix. But not every matrix is a rotational matrix. To turn a matrix into a rotational matrix, we need to do orthogonalization, such as using the Gram-Schmidt method. So our choice is to use quaternion.

2.1 Quaternion

A quaternion can be defined as \(q = [x, y, z, w]\), such that \(|q| = \sqrt{x^2 + y^2 + z^2 + w^2} = 1\).

Intuitively, it stands for a rotation around axis \((x, y, z)\) with angle \(\theta\), such that \(w = \cos(\theta/2)\) and \(x^2 + y^2 + z^2 = \sin^2(\theta/2)\).

A quaternion can be converted into a rotational matrix as:

\[
\begin{bmatrix}
 w^2 + x^2 - y^2 - z^2 & 2(xy + wz) & 2(xz - wy) \\
 2(xy - wz) & w^2 - x^2 + y^2 - z^2 & 2(yz + wx) \\
 2(xz + wy) & 2(yz - wx) & w^2 - x^2 - y^2 + z^2
\end{bmatrix}
\]  

(8)

**Quatation addition.**

\[
[v_1, s_1] + [v_2, s_2] = [v_1 + v_2, s_1 + s_2].
\]  

(9)

**Quatation-scalar multiplication.**

\[
s[v_1, s_1] = [sv_1 +, ss_1].
\]  

(10)

**Quatation-quaternion multiplication.**

\[
[v_1, s_1][v_2, s_2] = [s_1v_2 + s_2v_1 + v_1 \times v_2, s_1s_2 - v_1 \cdot v_2].
\]  

(11)

2.2 Definitions and Updates

Here we introduce some terms used in rotational motion and formulate the updates of the angular velocity and the quaternion.

**Inertia matrix.** In the local space, we can define the “rotational mass” of a rigid body using a matrix:

\[
I_{body} = \sum \left[ (p_i^T p_i - p_i p_i^T) \right].
\]  

(12)

in which \(I\) is the 3-by-3 identity matrix. The inertia matrix is not a constant and in the world space it changes to:

\[
I(t) = R(t) I_{body} R^T(t),
\]  

(13)

where \(R(t)\) is the rotational matrix of \(q(t)\).

**Torque.** The “rotational force” is the torque:

\[
\tau(t) = \sum (R(t)p_i \times f_i(t)).
\]  

(14)

If \(f_i(t)\) contains only the gravity force and the damping force, it can be proven that \(\tau(t) = 0\). In other words, we can ignore these two forces in \(f_i(t)\) when we calculate the torque. This is not surprising, because the gravity force and the damping force should not cause the object to spin.

**Angular velocity.** The angular velocity can be define as: \(\dot{\omega}(t) = [\omega_x, \omega_y, \omega_z]\). Here \(\omega_x, \omega_y, \omega_z\) gives the angular velocity axis and \(|\dot{\omega}|\) gives its speed magnitude.

The relationship between the angular velocity and the torque is very similar to \(v(t) = a(t) = \frac{\tau(t)}{I(t)}\):

\[
\dot{\omega}(t) = I^{-1}(t) \tau(t).
\]  

(15)
The relationship between the angular velocity and the time derivative of the quaternion is:

\[
q(t) = \frac{1}{2}[\tilde{\omega}(t), 0]q(t).
\]  

(16)

Here \([\tilde{\omega}(t), 0]\) is a new quaternion created from \(\tilde{\omega}(t)\). Note that the above equation is calculated using quaternion multiplication.

**Angular velocity and quaternion update.** Given \(\tilde{\omega}(t)\) and \(q(t)\), we first calculate the matrix \(R(t)\).
From \(R(t)\), we calculate inertia matrix \(I(t)\) and torque \(\bar{\tau}(t)\).

We then update the angular velocity as:

\[
\tilde{\omega}(t + \Delta t) = \tilde{\omega}(t) + \Delta t I^{-1}(t)\bar{\tau}(t),
\]

and update the quaternion as:

\[
q(t + \Delta t) = q(t) + \frac{\Delta t}{2}[\tilde{\omega}(t + \Delta t), 0]q(t).
\]

(18)

The last step is to re-normalize \(q(t + \Delta t)\), so that \(|q(t + \Delta t)| = 1\).

## 3 Collision Handling

In this section, we discuss how to handle collisions. To know if a vertex \(i\) is in collision with the ground plane \(y = 0\), we just need to test if the \(y\) component of \(x(t) + R(t)p_i\) is below zero.

In Unity, you can compute \(x(t) + R(t)p_i\) by simply calling \(\text{transform}\).\(\text{TransformPoint}(r_i)\).

### 3.1 Penalty force

Suppose we found that vertex \(i\) is in collision. We apply a penalty force on \(i\):

\[
f_i^{\text{penalty}} = [0, -k_p y_i(t), 0],
\]

(19)

in which \(k_p\) is a stiffness constant \((80000\) for example\) and \(y_i(t)\) is the \(y\) component of \(x(t) + R(t)p_i\). Intuitively, this force tries to push \(i\) out of the ground plane.

This force can directly be integrated into the total force and the total torque used in simulating linear and rotational motion.

### 3.2 Impulse method

The velocity of vertex \(i\) is: \(v_i(t) = \dot{x}_i(t) + \tilde{\omega}(t) \times (R(t)p_i)\). Let \(v_{0i}(t)\) be the \(y\) component of \(v_i(t)\). If \(v_{0i}(t) > 0\), the vertex is leaving the ground plane already so we do nothing.

If \(v_{0i}(t) < 0\), our goal is to apply an impulse to turn it into:

\[
-\mu v_{0i}(t),
\]

(20)
in which \(\mu\) is the restitution coefficient.

### The question is how large this impulse \(j\) should be?

According to Newton’s law, the impulse causes a sudden change to both linear velocity and angular velocity. The linear velocity is changed by:

\[
\Delta \mathbf{v} = \frac{1}{M} \mathbf{j}.
\]

(21)
The angular velocity is changed by:

\[
\Delta \tilde{\omega} = \Gamma^{-1} [r_i(t) \times j],
\]

(22)
in which \(r_i(t) = R(t)p_i\).

So the total velocity change at vertex \(i\) is:

\[
(\mathbf{v}_i(t) - \mathbf{v}_{0i}(t)) \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \frac{1}{M} \mathbf{j} + \Gamma^{-1} [r_i(t) \times j] \times r_i(t).
\]

(23)

If we formulate the cross product by a matrix:

\[
\mathbf{r} = \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix},
\]

(24)

we have \(r_i(t) \times j = \mathbf{r}^* j\). So we get:

\[
v_{0i}(t) \begin{bmatrix} 0 \\ -\mu - 1 \\ 0 \end{bmatrix} = \frac{1}{2\mu} \mathbf{r}^* j - \mathbf{r} \Gamma^{-1} [r_i(t) \times j]
\]

\[
= \frac{1}{2\mu} \mathbf{r} \Gamma^{-1} \mathbf{r} \mathbf{r}^* j = (\frac{1}{2\mu} \mathbf{I} - \mathbf{r} \Gamma^{-1} \mathbf{r}^*) j = \mathbf{K} j.
\]

(25)

We can then solve:

\[
j = v_{0i}(t) \mathbf{K}^{-1} \begin{bmatrix} 0 \\ -\mu - 1 \\ 0 \end{bmatrix}.
\]

(26)

Once we get \(j\), we add the velocity changes \(\Delta \mathbf{v}\) and \(\Delta \tilde{\omega}\) to the velocities accordingly.

### What if there are many colliding vertices? We detect all of them first, and then compute their average position. Use this average position as \(p_i\).

**Why do I see sliding?** Because we don’t have friction here. One solution is to use static friction. This means the velocity change at vertex \(i\) should be:

\[
\begin{bmatrix} 0 \\ -\mu v_{0i}(t) \\ 0 \end{bmatrix} = -\mathbf{v}_i(t),
\]

(27)

and the impulse \(j\) would be:

\[
j = \mathbf{K}^{-1} \begin{bmatrix} 0 \\ -\mu v_{0i}(t) \\ 0 \end{bmatrix} - \mathbf{v}_i(t).
\]

(28)

**Why do I see oscillation when the body rests on the floor?** Because it is never really at the rest. In each time step, it collides with the floor, get bounced back, then collides with the floor again sometime later. To solve this problem, we treat this contact case by examining \(v_{0i}(t)\). If \(|v_{0i}(t)|\) is small, we set the restitution \(\mu = 0\) to reduce this bouncing back artifact.

### Summary

Here is the summary of the impulse method.

**Algorithm 1 Update(x, v, q, \tilde{\omega})**

Compute the force \(f\) and the torque \(\tau\);
Update \(v\) and \(\tilde{\omega}\) using Equation 6 and 17;
Detect the vertices in collision and compute their average position \(r_i(t)\) at time \(t\);
Calculate the collision impulse \(j\) using Equation 28;
Apply additional changes to \(v\) and \(\tilde{\omega}\) using Equation 21 and 22;
Update \(x\) and \(q\) and apply them to the object in Unity.