Ruitian Yang
No. 3036410718

Task 1: Basic Rigid Body Simulator

Simulation Parameters

Below are the parameters used in the implementation:

1. **Particle Parameters**:
    - `particle_mass`: `1.0`
    - `gravity`: `[0.0, -9.8, 0.0]`
2. **Body Parameters**:
    - `initial_velocity`: `[4.0, 2.0, 3.0]`
    - `initial_angular_velocity`: `[0.0, 100.0, 0.0]`
3. **Collision Parameters**:
    - `collision_stiffness`: `1e4`
    - `velocity_damping_stiffness`: `1e3`
    - `friction_stiffness`: `0.1`
4. **Integration Parameters**:
    - `dt`: `1e-3` (time step for simulation)

Implementation Overview

The rigid body simulator has been implemented successfully by completing the **7 required TODOs** in the provided code. Below is a summary of how each step was addressed:

1. **TODO 1: Compute the Inertia Tensor**
   The inertia tensor for the rigid body was computed using the following formula:
   ```
   inertia += particle_mass * (r.dot(r) * I - r.outer_product(r))
   ```
   Here, `r` is the relative position of each particle from the center of mass, and `I` is the identity matrix. The inverse of the inertia tensor was also computed and stored for later use.

2. **TODO 2: Gravity Force**

   Gravity was applied to each particle by computing the gravitational force as:

   ```
   particle_force[i] = gravity * particle_mass
   ```

3. **TODO 3: Compute the Force for Rigid Body**

   The total force acting on the rigid body was calculated by summing up all the forces acting on individual particles.

4. **TODO 4: Compute the Torque for Rigid Body**

   The torque on the rigid body was computed using the cross product of the relative position vector (`r_i`) and the force acting on each particle:

   ```
   body_torque += ti.math.cross(r_i, particle_force[i])
   ```

5. **TODO 5: Update the Center of Mass Position and Velocity**

   The center of mass position and velocity were updated using the force and acceleration:

   ```
   body_velocity[None] += (body_force / body_mass[None]) * dt
   body_cm_position[None] += body_velocity[None] * dt
   ```

6. **TODO 6: Update the Rotation Quaternion**

   The rotation quaternion was updated using the angular velocity and the following formula:

   ```
   d_q = 0.5 * quaternion_multiplication([0, omega[0], omega[1], \
   omega[2]], body_rotation_quaternion[None])
   body_rotation_quaternion[None] += d_q * dt
   ```

   The quaternion was normalized to prevent numerical errors.

7. **TODO 7: Update Angular Momentum, Inertia Tensor, and Angular Velocity**

   The angular momentum was updated using the torque:

   ```
   body_angular_momentum[None] += body_torque * dt
   ```

   The current inverse inertia tensor in world space was calculated as:

```
world_inverse_inertia = body_rotation[None] @ \
body_origin_inverse_inertia[None] @ body_rotation[None].transpose()
```

Finally, the angular velocity was updated using the inertia tensor and angular momentum:

```
body_angular_velocity[None] = world_inverse_inertia \
@ body_angular_momentum[None]
```

The particle positions and velocities were also updated using the rigid body's updated center of mass and angular velocity.

Task 2: Velocity Damping

Implementation of Velocity Damping

To improve the realism of the simulation, velocity damping was implemented to simulate energy loss due to air resistance or internal material deformations. This was achieved by applying a damping force proportional to the particle's velocity, effectively reducing oscillations and stabilizing the system over time. Below is an explanation of the implementation:

1. **Damping Force Formula**
   The damping force is calculated as:

```
F_damping = -c \* v
```

where:

- `c` is the damping coefficient (`velocity_damping_stiffness`), set to `1.0` in the simulation.
- `v` is the velocity of the particle.

This force opposes the direction of velocity and reduces the overall energy of the system over time.

2. **Code Implementation**
   In the `substep()` function, the damping force was added to the particle's force calculation after gravity. The updated force computation looks as follows:

```
particle_force[i] = gravity * particle_mass
particle_force[i] -= velocity_damping_stiffness * particle_velocities[i]
```

3. **Effect on Simulation**
   - The damping force gradually slows down the particles in motion, mimicking the effects of air resistance or internal friction.
   - It prevents particles from oscillating indefinitely and helps the system converge to a stable state over time.

Advantages of Velocity Damping

- **Stability**: Damping reduces high-frequency oscillations, making the simulation more stable.
- **Realism**: Simulates the natural dissipation of energy in physical systems due to friction or drag.
- **Control**: The damping coefficient (`velocity_damping_stiffness`) provides an intuitive parameter to control the rate of energy loss in the system.

Parameter Used

- **Damping Coefficient (`velocity_damping_stiffness`)**: Set to `1.0` for this simulation, providing a balance between energy dissipation and stability.

With the addition of velocity damping, the simulation is now better suited to mimic real-world physical behavior by accounting for energy dissipation over time.