CS171.01 Final Project:
Massive Rigid-Body Simulation

NAME: 丁弘毅 王鸿润 刘放勋
STUDENT ID: 2020533039 2020533102 2020533047
EMAIL: DINGHY1|WANGHR|LIUFX@SHANGHAITECH.EDU.CN

1 INTRODUCTION & WORKLOAD

In this project, we are going to simulate massive rigid body. The workload can be divided mainly into three parts: transformation over time, collision detection and collision response. These parts are finished by 王鸿润, 丁弘毅 and 刘放勋 respectively.
The whole pipeline is that in each fixed time interval, we simulate the whole scene for a certain number of times, including updating positions & velocities and coping with collisions. In each single simulation, we first update the position, velocity and angular velocity of a rigid body according to its acceleration, induced by forces from gravity, collision and friction. Then we check whether there are two of the objects in the scene collide with each other, and if so, we get the collision position, the normal and the signed distance. At last, in the collision response, we tear the two objects apart, and change the velocity & angular velocity of them according to physical formulas.
Here are some useful links:

- GAMES103 Lecture 3,4,9

2 BRIEF INTRODUCTION OF THE PAPER'S WORK

In the time background of the paper, many rigid body physics algorithms were slow, used too much memory, were difficult to implement, or had other nasty limitations. The author of *Iterative Dynamics with Temporal Coherence* raised three ideas:

- Use an approximate contact model that is easy to solve.
- Use a sloppy but fast constraint solver.
- Clean up the solution over several frames.

The author used the following toolkits in his implementation:

- Contact point calculator.
- Rigid bodies, constraints, and Jacobians.
- Gauss-Seidel constraint solver and simple integrator.
- Contact cache.

The key procedure of his algorithm is shown below:
In each time stepping:

- 1. Generate contact points.
- 2. Initialize contact forces λ using a contact cache (generated in the previous step).
- 3. Compute the Jacobian $\mathbf{J}$ for non-penetration and friction constraints.
- 4. Form an equation for λ.
- 5. Use a Gauss-Seidel solver to refine λ.
- 6. Compute new velocities v and ω using λ.
- 7. Compute new positions x and q from v and ω.
- 8. Store λ in the contact cache.
- 9. Go to step 1.

His algorithm can be summarized as:

- Compute the collision point.
- Apply external force (such as gravity).
- Apply impulse.
- Update the positions.
- Go back to loop.

3 IMPLEMENTATION DETAILS

3.1 Transform over time

For each object, it has its own velocity and angular velocity, acceleration and angular acceleration.In simulation,we update the status of every object ,compute the position and normal.
For velocity,

$$velocity+ = acceleration \times Ddeltat, \quad position+ = v \times \Delta t.$$

For angular velocity,

$$\omega+ = angularacceleration \times \Delta t$$

and the Quaternion of rotation is $q = q + \{0, \frac{\delta t}{2}\omega\} \times q$ and normalize the Quaternion.
Then apply the transform function to change positon.

3.2 Collision Detection

To judge whether two rigid-body meshes A and B collide with each other, the universal idea is to traverse each vertex a of A, if there is some a collides with B, then there is a collision and

we need to respond to it. If there are multiple a's colliding with B, we can just view it as a collision at the average position of collision with average normal and average distance. But actually if time slice is small enough, there will only be one point colliding with some other objects each time.
As to how to know whether a point a collides with B, we only need to know the sdf (signed distance function) of B on point a. It is defined as the maximum signed distance from any point of B to a. For uniseral triangle mesh, we need to traverse each triangle, and calculate the distance from a to the plane of triangle, and finally take the maximum value, which turns out to be the sdf of B on a.
However, the universal way might be costy and slow, which wastes plenty of time on non-intersect objects and verteces. First, instead of scanning every pair of objects, we can skip those pairs that are too far away with each other, either by Spatial Hashing or BVH (Bounding Volumn Hierarchies). Second, we don't need to calculate the sdf of B on all the vertices a. Since most of those vertices won't collide with B, we can skip those vertices with some data structures. Because the structure of a rigid-body is stable, we can construct a BVH for each object, then skip the vertices whose AABB does not intersect with the AABB of B.
At last, if objects are special geometries, we can make special optimizations for them. For example, for spheres, if we want to know the sdf of sphere O on P, we only need to calculate $|OP| - r$. Similarly, if we want to know whether two spheres O_1 and O_2 intersects, we only need to judge whether $|O_1O_2| - r1 - r_2 > 0$. This is of great improvement than the uniseral traverse of vertices and mesh triangles. The data sent from detector to responder is defined as follow:

```
struct Interaction{
  Float dis;      // signed distance
  Vec3 position;  // position of vertex a that collides
  Vec3 normal;    // the normal of object B's colliding plane
};
```

3.3 Collision Response

First, we implement a simple form of collision response. If the velocity v of current object is pointing into the collided object, we check the dot product of $\mathbf{v}$ and normal $\mathbf{n}$ of the collision point: $t = \mathbf{v} \cdot \mathbf{n}$. If $t < 0$, we reverse both $\mathbf{v}$ and t. Then we use the following way to update the velocity:

$$\mathbf{v} \leftarrow 2t\mathbf{n} - \mathbf{v}$$

To make the resting contact stable, we judge that when the velocity is too small, stop updating position of the object.

Then we look into a more advanced collision response. We choose the impulse method. Its basic idea is to use the velocity change of the collision point to get the impulse applied on the object, then use the impulse to update the velocity of the whole object. We implement the impulse method in the following four steps.

Step 1: Judge whether the update is needed
In the previous collision detection phase, we have detected the collision point. Then we need to judge whether the velocity $\mathbf{v}_i$ of the collision point is pointing to the inside of the collided object. If not, then we don't need to do the update as the velocity is already outward.

Step 2: Compute the new velocity $\mathbf{v}_i^{new}$ of the collision point i
Denote the normal of the collision point to be $\mathbf{N}$, pointing to the outside of the collided object. First we decompose the old velocity $\mathbf{v}_i$ into normal velocity $\mathbf{v}_{\mathrm{N},i}$ and tangential velocity $\mathbf{v}_{\mathrm{T},i}$:

$$\mathbf{v}_{\mathrm{N},i} \leftarrow (\mathbf{v}_i \cdot \mathbf{N})\mathbf{N}$$

$$\mathbf{v}_{\mathrm{T},i} \leftarrow \mathbf{v}_i - \mathbf{v}_{\mathrm{T},i}$$

We introduce two coefficients $\mu_\mathrm{N}, \mu_\mathrm{T} \in (0, 1)$ to update the two decomposed velocities separately. We wish our object to go away from the collided object, so the normal velocity should be reversed, and by conservation of energy, the out normal velocity should be no larger than the in normal velocity, and may decay in energy. So we update the normal velocity by the following:

$$\mathbf{v}_{\mathrm{N},i}^{new} \leftarrow -\mu_\mathrm{N}\mathbf{v}_{\mathrm{N},i}$$

Due to the existence of friction, the tangential velocity should also decay in energy. We update the tangential velocity by the following:

$$\mathbf{v}_{\mathrm{T},i}^{new} \leftarrow a\mathbf{v}_{\mathrm{T},i}$$

The computation of a is needed. By Amontons-Coulomb Friction Laws, we know

$$F_f \le \mu_\mathrm{T} F_N,$$

where μ_T is the friction coefficient, then

$$||\mathbf{v}_{\mathrm{T},i}^{new} - \mathbf{v}_{\mathrm{T},i}|| \le \mu_\mathrm{T}||\mathbf{v}_{\mathrm{N},i}^{new} - \mathbf{v}_{\mathrm{N},i}||$$

$$(1-a)||\mathbf{v}_{\mathrm{T},i}|| \le \mu_\mathrm{T}(1+\mu_\mathrm{N})||\mathbf{v}_{\mathrm{N},i}||$$

$$a \le 1 - \frac{\mu_\mathrm{T}(1+\mu_\mathrm{N})||\mathbf{v}_{\mathrm{N},i}||}{||\mathbf{v}_{\mathrm{T},i}||}$$

In our simulation, we take the equals sign. a should also satisfies $a \ge 0$, so the computation of a is:

$$a \leftarrow \max(1 - \frac{\mu_\mathrm{T}(1+\mu_\mathrm{N})||\mathbf{v}_{\mathrm{N},i}||}{||\mathbf{v}_{\mathrm{T},i}||}, 0)$$

$$\mathbf{r} \times \mathbf{q} = \begin{bmatrix} r_y q_z - r_z q_y \\ r_z q_x - r_x q_z \\ r_x q_y - r_y q_x \end{bmatrix} = \begin{bmatrix} 0 & -r_z & r_y \\ r_z & 0 & -r_x \\ -r_y & r_x & 0 \end{bmatrix} \begin{bmatrix} q_x \\ q_y \\ q_z \end{bmatrix} = \mathbf{r}^* \mathbf{q}$$

Fig. 1. convertion of cross product to matrix product

Now we have the normal and tangential velocity of the new velocity, then the new velocity is:

$$\mathbf{v}_i^{new} \leftarrow \mathbf{v}_{\mathrm{N},i} + \mathbf{v}_{\mathrm{T},i}$$

Step 3: Compute the impulse j

Assume the impulse is applied on the collision point, it will change the velocity and the angular velocity of the object:

$$\mathbf{v}^{new} \leftarrow \mathbf{v} + \frac{1}{M}\mathbf{j}$$

$$\omega^{new} \leftarrow \omega + \mathbf{I}^{-1}(\mathbf{R}\mathbf{r}_i \times \mathbf{j})$$

where $\mathbf{I}$ is the inertia, $\mathbf{R}$ is the rotation matrix and $\mathbf{r}$ is the local coordinate of the collision point. $(\mathbf{R}\mathbf{r}_i \times \mathbf{j})$ is the torque induced by $\mathbf{j}$.

Then we take the $\mathbf{v}^{new}, \omega^{new}$ into following equation:

$$\begin{aligned} \mathbf{v}_i^{new} &= \mathbf{v}^{new} + \omega^{new} \times \mathbf{R}\mathbf{r}_i \\ &= \mathbf{v} + \frac{1}{M}\mathbf{j} + (\omega + \mathbf{I}^{-1}(\mathbf{R}\mathbf{r}_i \times \mathbf{j})) \times \mathbf{R}\mathbf{r}_i \\ &= \mathbf{v}_i + \frac{1}{M}\mathbf{j} + (\mathbf{I}^{-1}(\mathbf{R}\mathbf{r}_i \times \mathbf{j})) \times \mathbf{R}\mathbf{r}_i \\ &= \mathbf{v}_i + \frac{1}{M}\mathbf{j} - (\mathbf{R}\mathbf{r}_i) \times (\mathbf{I}^{-1}(\mathbf{R}\mathbf{r}_i \times \mathbf{j})) \end{aligned}$$

We convert the cross product $mathbfr\times$ into a matrix product $mathbfr*$ in the way shown in Fig.1. Then we can convert the above equation to the following:

$$\mathbf{v}_i^{new} = \mathbf{v}_i + \frac{1}{M}\mathbf{j} - (\mathbf{R}\mathbf{r}_i) * \mathbf{I}^{-1}(\mathbf{R}\mathbf{r}_i) * \mathbf{j}$$

So we can get:

$$\mathbf{v}_i^{new} - \mathbf{v}_i = \mathbf{K}\mathbf{j}$$

where

$$\mathbf{K} \leftarrow \frac{1}{M}\mathbf{1} - (\mathbf{R}\mathbf{r}_i) * \mathbf{I}^{-1}(\mathbf{R}\mathbf{r}_i)*$$

Now we can get the impulse $\mathbf{j}$:

$$\mathbf{j} \leftarrow \mathbf{K}^{-1}(\mathbf{v}_i^{new} - \mathbf{v}_i)$$

Step 4: Update v and ω

Lastly, we need to use the impulse to update $\mathbf{v}$ and ω:

$$\mathbf{v} \leftarrow \mathbf{v} + \frac{1}{M}\mathbf{j}$$

$$\omega \leftarrow \omega + \mathbf{I}^{-1}(\mathbf{R}\mathbf{r}_i \times \mathbf{j})$$

Now the collision response using impulse method has finished.

Disappointingly, although we do the impulse method implementation following the above procedure, the simulaiton could not work well. We guess that it may be due to some difference on settings. At last, we use the first implementation to deal with collision response, but still retain the impulse method implementation in our code.

4 RESULTS