Computer Animation Homework 1

112550184 陳璽安

1. Implementation

jelly.cpp

a. void Jelly::initializateSpring():

Traverse the node in x direction, y direction and z direction respectively, and compute the spring length (length between the nodes) and connect the nodes with the spring. Connecting the nodes in the same plane with structural spring and blending spring, and connect the nodes in the different planes with shear spring. The code below is the implementation of connect structural spring on the x direction.

```
//x-direction
for (int k = 0; k < particleNumPerEdge; k++) {
    for (int j = 0; j < particleNumPerEdge; j++) {
        for (int i = 0; i < particleNumPerEdge - 1; i++) {
            int iParticleID = i * particleNumPerFace + j * particleNumPerEdge + k;
            int iNeighborID = (i+1) * particleNumPerFace + j * particleNumPerEdge + k;
            Eigen::Vector3f SpringStartPos = particles[iParticleID].getPosition();
            Eigen::Vector3f SpringEndPos = particles[iNeighborID].getPosition();
            Eigen::Vector3f Length = SpringStartPos - SpringEndPos;
            float absLength = sqrt(Xx: Length[0] * Length[0] + Length[1] * Length[1] + Length[2] * Length[2]);

            springs.push_back(Val: Spring(iParticleID, iNeighborID, absLength, springCoefStruct, damperCoefStruct,
                                          Spring::SpringType::STRUCT));
            struct_num++;
        }
    }
}
```

b. Eigen::Vector3f Jelly::computeSpringForce(…)

Compute the vector, length between the nodes and get the unit vector direction (vector divides length). And compute spring force by Hook's Law : spring force= -springCoef * (length-restLength) *unit vector direction.

```
    Eigen::Vector3f vector = positionB - positionA;  // vector b to a

    float len = vector.norm();

    if (len < 1e-6) {
        return Eigen::Vector3f::Zero();  // if spring len very small return 0
    }

    Eigen::Vector3f dir = vector / len;  // unit vector
    float x = len - restLength;

    Eigen::Vector3f spring_force = springCoef * x * dir;  // f=-kx

    return spring_force;
}
```

c. Eigen::Vector3f Jelly::computeDamperForce(…)

Calculate the unit vector along the string and compute the relative velocity along the spring by inner production. Then compute damper force by -dampCoef * (relative velocity along the spring) * spring unit vector.

```
Eigen::Vector3f Jelly::computeDamperForce(const Eigen::Vector3f &positionA, const Eigen::Vector3f &positionB,
                                          const Eigen::Vector3f &velocityA, const Eigen::Vector3f &velocityB,
                                          const float damperCoef) {
    // TODO#2-2: Compute damper force given the two positions and the two velocities of the spring.
    //   1. Review "particles.pptx" from p.9 - p.13
    //   2. The sample below just set damper force to zero

    Eigen::Vector3f vector = positionA - positionB;
    float len = vector.norm();

    // if len is very small return 0
    if (len < 1e-6) {
        return Eigen::Vector3f::Zero();
    }

    // normalize the vector
    vector /= len;

    Eigen::Vector3f vba = velocityA - velocityB;
    float vba_along_spring = vba.dot(other: vector);

    Eigen::Vector3f damper_force = -damperCoef * vba_along_spring * vector;
    return damper_force;
}
```

d. void Jelly::computeInternalForce()

For each spring, we get position and velocity of its start node and ended node as parameters to plug in the functions. Compute spring force and damper force by function computeSpringForce and computeDamperForce, and add them together to get the internal force. Then store the internal force of the nodes by .addForce() function.

```
for (int i = 0; i < springs.size(); i++) {
    int start = springs[i].getSpringStartID();
    int end = springs[i].getSpringEndID();

    Eigen::Vector3f xa = particles[start].getPosition();
    Eigen::Vector3f xb = particles[start].getPosition();
    Eigen::Vector3f va = particles[start].getVelocity();
    Eigen::Vector3f vb = particles[start].getVelocity();

    float springCoef = springs[i].getSpringCoef();
    float damperCoef = springs[i].getDamperCoef();
    float restLength = springs[i].getSpringRestLength();

    Eigen::Vector3f f_spring = computeSpringForce(positionA: xa, positionB: xb, springCoef, restLength);

    Eigen::Vector3f f_damper = computeDamperForce(positionA: xa, positionB: xb, velocityA: va, velocityB: vb, damperCoef);

    Eigen::Vector3f f_internal = f_spring + f_damper;

    particles[start].addForce(_force: f_internal);
    particles[end].addForce(_force: -f_internal);
}
```

tarrain.cpp

a. void PlaneTerrain::handleCollision(...)

For each particle, get its position and velocity.

```
for (int i = 0; i < jelly.getParticleNum(); i++) {
    Particle& particle = jelly.getParticle( particleIdx: i);

    Eigen::Vector3f x = particle.getPosition();
    Eigen::Vector3f v = particle.getVelocity();
```

Calaulate the distance between particle and the plane by inner product of particle's position and the normal vector of plane.

If distance < eEPSILON, collision occurs.

```
float dis = normal.dot( other: x);

if (dis < eEPSILON) {
```

Compute particle's velocity that is perpendicular to the plane, and only handle particles that moves toward the plane.

```
float v_along_nor = v.dot( other: normal);
if (v_along_nor < 0) {
```

Get velocity's tangent component (v_T) and normal component (v_N), and compute velocity after collision by $v' = -k_r * v_N + v_T$.

Set the velocity of particle be the new velocity.

```
Eigen::Vector3f vn = v_along_nor * normal;
Eigen::Vector3f vt = v - vn;
Eigen::Vector3f reflect_v = vt - coefResist * vn;
particle.setVelocity(reflect_v);
```

Position correction to prevent penetration.

```
if (dis < 0) {
    particle.addPosition(-dis * normal);
}
```

Check whether |N·(x-p)|<epsilon, if yes apply contact force and friction.

Is there's a force that pushes particle toward the wall, compute contact force by fc = -(N·f)N.

```
if (std::abs(dis) < eEPSILON) {
    Eigen::Vector3f force = particle.getForce();
    float force_nor = force.dot(normal);

    //force pushes particle into the plane
    if (force_nor < 0) {
        Eigen::Vector3f contactForce = -force_nor * normal;
        ////fc = -(N·f)N
```

If velocity tangent component is greater than 1e-6, apply friction force.

Compute friction force by f = -kf(-N·f)vt, and add the force to the particle.

```
////fc = -(N·f)N
Eigen::Vector3f vt = v - v.dot(normal) * normal;
if (vt.norm() > 1e-6) {
    Eigen::Vector3f friction = -coefFriction * (-force_nor) * vt.normalized();
    // f = -kf(-N·f)vt
    particle.addForce(contactForce + friction);
}
else {
    particle.addForce(contactForce);
}
```

b. void ElevatorTerrain::handleCollision(...)

For each particle, check whether the particle is in the range of elevator.

(Since the elevator is one the xz plane, we need to check x coordinate and z coordinate of the particle.)

```
for (int i = 0; i < jelly.getParticleNum(); i++) {
    Particle& particle = jelly.getParticle(i);

    Eigen::Vector3f x = particle.getPosition();
    Eigen::Vector3f v = particle.getVelocity();

    float halfLen = 2.5f;
    //half width of elevator
    if (std::abs(x[0]) <= halfLen && std::abs(x[2]) <= halfLen) {
```

Check whether N·(x-p) < ε .

Compute the relative velocity between particle and elevator, and get the velocity in normal direction.

```
float dis = (x - position).dot(normal);
// N·(x-p) < ε
if (dis < eEPSILON) {
    Eigen::Vector3f relative_v = v - velocity;
    //relative velocity of particle and evaletor

    float nor_v = relative_v.dot(normal);
```

Compute velocity after collision (relative to the elevator's velocity) by

$v' = -k_r * v_N + v_T$

And get the absolute velocity of particle by adding the velocity of elevator.

Change the velocity of particle to the computed result.

```
if (nor_v < 0) {
    //particle is moving toward the elevator
    Eigen::Vector3f new_v = relative_v - (1 + coefResist) *nor_v * normal;

    new_v += velocity;//add  velocity of elevator to get the absolute velocity
    particle.setVelocity(new_v);

    if (dis < 0) {
        Eigen::Vector3f correction = -dis * normal;
        particle.addPosition(correction);
    }
}
```

If N·(x-p) < ε , apply contact force and friction.

If there's a force push particle toward plane, compute contact force

fc = -(N·f)N

```
if (std::abs(dis) < eEPSILON) {
    //apply contact force

    Eigen::Vector3f force = particle.getForce();

    float force_nor = force.dot(normal);

    //a force push particle toward the surface (N·f < 0)
    if (force_nor < 0) {
        Eigen::Vector3f contactForce = -force_nor * normal;//fc = -(N·f)N
```

Compute tangent component of particle's velocity, and calculate friction as f= -kf(-N·f)vt

And apply the force to the particle.

```
Eigen::Vector3f normalComponent = nor_v * normal;
Eigen::Vector3f tangentComponent = relative_v - normalComponent;

Eigen::Vector3f friction;
if (tangentComponent.norm() > 1e-6) {
    friction = -coefFriction * (-force_nor) * tangentComponent.normalized();
    // f = -kf(-N·f)vt
}
else {
    friction = Eigen::Vector3f::Zero();
}

particle.addForce(contactForce + friction);
```

Integrator.cpp

a. void ExplicitEulerIntegrator::integrate(…)

Get jelly object and get delta time.

```
Jelly* jelly = particleSystem.getJellyPointer(n:0);
float dt = particleSystem.deltaTime;
```

For each particle in jelly, get their position and velocity.

```
for (int i = 0; i < jelly->getParticleNum(); i++) {
    Particle& particle = jelly->getParticle(particleIdx:i);

    Eigen::Vector3f x = particle.getPosition();
    Eigen::Vector3f v = particle.getVelocity();
```

Compute new position and velocity in t = t + delta_t by explicit Euler method (x'=x+v*t, v'=v+at)

```
//x(t+dt) = x(t) + v(t) * dt
x += v * dt;
particle.setPosition(_position:x);

Eigen::Vector3f f = particle.getForce();
float m = particle.getMass();
// v(t+dt) = v(t) + a(t) * dt
v += (f / m) * dt;
particle.setVelocity(_velocity:v);
```

Clear force

```
//clear force
particle.setForce(_force:Eigen::Vector3f::Zero());
```

If it's elevator scene, update elevator's position and velocity by using explicit Euler method. (Like we did in updating particle's position and velocity)

```
if (particleSystem.sceneIdx == 1) {
    Eigen::Vector3f x = particleSystem.elevatorTerrain->getPosition();
    Eigen::Vector3f v = particleSystem.elevatorTerrain->getVelocity();
    Eigen::Vector3f f = particleSystem.elevatorTerrain->getForce();
    float m = particleSystem.elevatorTerrain->getMass();

    // v(t+dt) = v(t) + a(t) * dt
    v += (f / m) * dt;
    particleSystem.elevatorTerrain->setVelocity(_velocity:v);

    // x(t+dt) = x(t) + v(t) * dt
    x += v * dt;
    particleSystem.elevatorTerrain->setPosition(_position:x);

    // 清除力
    particleSystem.elevatorTerrain->setForce(_force:Eigen::Vector3f::Zero());
}
```

b. void ImplicitEulerIntegrator::integrate(…)

Get jelly object and delta t, and create vectors to store original states of the jelly (including position, velocity and force).

```
Jelly* jelly = particleSystem.getJellyPointer(0);
float dt = particleSystem.deltaTime;

std::vector<Eigen::Vector3f> x;
std::vector<Eigen::Vector3f> v;
std::vector<Eigen::Vector3f> f;
int n = jelly->getParticleNum();

x.reserve(n);
v.reserve(n);
f.reserve(n);
```

For each particle in jelly, store its original state.

```
// store original states of particles
for (int i = 0; i < n; i++) {
    Particle& p = jelly->getParticle(i);
    x.push_back(p.getPosition());
    v.push_back(p.getVelocity());
    f.push_back(p.getForce());
}
```

Perform the explicit Euler's method and we will get the force in the next time slot (t = t+ delta_t) applied on the particle by using these updated parameters later.

```
// perform explicit euler method first
for (int i = 0; i < n; i++) {
    Particle& p = jelly->getParticle(i);
    Eigen::Vector3f xi = x[i];
    Eigen::Vector3f vi = v[i];
    Eigen::Vector3f fi = f[i];
    float m = p.getMass();

    xi += vi * dt;
    vi += (fi / m) * dt;

    p.setPosition(xi);
    p.setVelocity(vi);
    p.setForce(Eigen::Vector3f::Zero());
}
```

If its elevator scene, store the original elevator state, and perform the explicit Euler's method to get the next state.

```
int elevator = particleSystem.elevatorCounter;
Eigen::Vector3f ex, ev, ef, ex1, ev1, ef1;
if (particleSystem.sceneIdx == 1) {
    ex = particleSystem.elevatorTerrain->getPosition();
    ev = particleSystem.elevatorTerrain->getVelocity();
    ef = particleSystem.elevatorTerrain->getForce();
    float em = particleSystem.elevatorTerrain->getMass();

    // explicit euler method
    ex1 = ex + ev * dt;
    ev1 = ev + (ef / em) * dt;
    particleSystem.elevatorTerrain->setPosition(ex1);
    particleSystem.elevatorTerrain->setVelocity(ev1);
    particleSystem.elevatorTerrain->setForce(Eigen::Vector3f::Zero());
}
```

Compute the force applied on elevator and jelly in the time = t+delta_t by the parameters we get by using explicit Euler's method.

```
if (particleSystem.sceneIdx == 1) particleSystem.computeElevatorForce();
particleSystem.computeJellyForce(*jelly);
```

Perform implicit Euler's method. Calaulate a_{i+1} by f_i+1/m, and compute v_{i+1} by a_{i+1} * t + v_i. Compute x_{i+1} by x_i + v_{i+1} * t.

Eventually, update the new state of the particle.

```
// perform implicit method
for (int i = 0; i < n; i++) {
    Particle& p = jelly->getParticle(i);
    Eigen::Vector3f f1 = p.getForce();
    float m = p.getMass();

    Eigen::Vector3f v1 = v[i] + (f1 / m) * dt;  // vi+1 = vi + ai+1 * t
    Eigen::Vector3f x1 = x[i] + v1 * dt;        // xi+1 = xi + vi+1 * t

    p.setPosition(x1);
    p.setVelocity(v1);
    p.setForce(Eigen::Vector3f::Zero());
}
```

If it's elevator scene, update elevator's state by using implicit Euler's method like how we do in updating particles' state.

```
if (particleSystem.sceneIdx == 1) {
    float em = particleSystem.elevatorTerrain->getMass();
    Eigen::Vector3f ef1 = particleSystem.elevatorTerrain->getForce();
    Eigen::Vector3f ev1 = ev + (ef1 / em) * dt;
    Eigen::Vector3f ex1 = ex + ev1 * dt;
    particleSystem.elevatorTerrain->setPosition(ex1);
    particleSystem.elevatorTerrain->setVelocity(ev1);
    particleSystem.elevatorTerrain->setForce(Eigen::Vector3f::Zero());
    particleSystem.elevatorCounter = elevator;
}
```

c. void MidpointEulerIntegrator::integrate(…)

Get jelly object and store the initial state of particles.

```
Jelly* jelly = particleSystem.getJellyPointer(0);
float dt = particleSystem.deltaTime;

std::vector<Eigen::Vector3f> x;
std::vector<Eigen::Vector3f> v;
std::vector<Eigen::Vector3f> f;
int n = jelly->getParticleNum();

x.reserve(n);
v.reserve(n);
f.reserve(n);

// store original states of particles
for (int i = 0; i < n; i++) {
    Particle& p = jelly->getParticle(i);
    x.push_back(p.getPosition());
    v.push_back(p.getVelocity());
    f.push_back(p.getForce());
}
```

Compute particle state at t = t + dt/2 by explicit Euler's method.

```
float halft = dt / 2.0f;
//perform explicit euler method in t + delta t/2
for (int i = 0; i < n; i++) {
    Particle& p = jelly->getParticle(i);
    Eigen::Vector3f xi = x[i];
    Eigen::Vector3f vi = v[i];
    Eigen::Vector3f fi = f[i];
    float m = p.getMass();

    xi += vi * halft;
    vi += (fi / m) * halft;

    p.setPosition(xi);
    p.setVelocity(vi);
    p.setForce(Eigen::Vector3f::Zero());
}
```

If it's elevator scene, store initial state of elevator and compute elevator's state at t = t + dt/2 by Euler's method.

```
int elevator = particleSystem.elevatorCounter;
Eigen::Vector3f ex, ev, ef, ex1, ev1, ef1;
if (particleSystem.sceneIdx == 1) {
    ex = particleSystem.elevatorTerrain->getPosition();
    ev = particleSystem.elevatorTerrain->getVelocity();
    ef = particleSystem.elevatorTerrain->getForce();
    float em = particleSystem.elevatorTerrain->getMass();

    // explicit euler method
    ex1 = ex + ev * halft;
    ev1 = ev + (ef / em) * halft;
    particleSystem.elevatorTerrain->setPosition(ex1);
    particleSystem.elevatorTerrain->setVelocity(ev1);
    particleSystem.elevatorTerrain->setForce(Eigen::Vector3f::Zero());
}
```

Compute force applied on elevator and jelly at t = t + dt/2

```
// jelly force at t=t+dt/2
if (particleSystem.sceneIdx == 1) particleSystem.computeElevatorForce();
particleSystem.computeJellyForce(*jelly);
```

Perform midpoint method to calculate the new state of the particle.

x(t+dt) = x(t) + v(t+dt/2) * dt

v(t + dt) = v(t) + a(t + dt / 2) * dt

Update its position and velocity by using setPosition and setVelocity, and clear the force.

```
// perform midpoint method
for (int i = 0; i < n; i++) {
    Particle& p = jelly->getParticle(i);
    Eigen::Vector3f f_mid = p.getForce();
    Eigen::Vector3f v_mid = p.getVelocity();
    float m = p.getMass();

    Eigen::Vector3f x1 = x[i] + v_mid * dt;
    Eigen::Vector3f v1 = v[i] + (f_mid / m) * dt;
    //x(t+dt) = x(t) + v(t+dt/2) * dt
   // v(t + dt) = v(t) + a(t + dt / 2) * dt

    p.setPosition(x1);
    p.setVelocity(v1);
    p.setForce(Eigen::Vector3f::Zero());
}
```

Apply midpoint method to calculate elevator's new state, and remember that we need to restore the elevator counter in the end.

```
if (particleSystem.sceneIdx == 1) {
    float em = particleSystem.elevatorTerrain->getMass();
    Eigen::Vector3f ef1 = particleSystem.elevatorTerrain->getForce();
    ex1 = ex + ev1 * dt;
    ev1 = ev + (ef1 / em) * dt;
    particleSystem.elevatorTerrain->setPosition(ex1);
    particleSystem.elevatorTerrain->setVelocity(ev1);
    particleSystem.elevatorTerrain->setForce(Eigen::Vector3f::Zero());
    particleSystem.elevatorCounter = elevator;
}
```

d. void MidpointEulerIntegrator::integrate(…)

Get the jelly object and store particles' initial position and velocity.

```
Jelly* jelly = particleSystem.getJellyPointer(n:0);
float dt = particleSystem.deltaTime;

std::vector<Eigen::Vector3f> x;
std::vector<Eigen::Vector3f> v;
int n = jelly->getParticleNum();

x.reserve(_Newcapacity:n);
v.reserve(_Newcapacity:n);

// store original states of particles
for (int i = 0; i < n; i++) {
    Particle& p = jelly->getParticle(particleIdx:i);
    x.push_back(_Val:p.getPosition());
    v.push_back(_Val:p.getVelocity());
    p.setForce(_force:Eigen::Vector3f::Zero());
}
```

Store elevator's initial state. Create a vector to store k value for particles, and set the force applied on particles to be zero to avoid force accumulation from previous computations.

```
int eleCounter = particleSystem.elevatorCounter;
Eigen::Vector3f ex = particleSystem.elevatorTerrain->getPosition();
Eigen::Vector3f ev = particleSystem.elevatorTerrain->getVelocity();

// k value for each particle
std::vector<std::vector<StateStep> > k(4, std::vector<StateStep>(n));

for (int i = 0; i < n; i++) {
    jelly->getParticle(particleIdx: i).setForce(_force: Eigen::Vector3f::Zero());
    // reset the force to aviod force accumulation
}
```

Compute force applied on elevator and particles.

```
if (particleSystem.sceneIdx == 1) {
    particleSystem.computeElevatorForce();
}
particleSystem.computeAllForce();
```

Compute k1= $hf(x_0, t_0)$

```
// k1 = hf(x0, t0)
for (int i = 0; i < n; i++) {
    Particle& p = jelly->getParticle(particleIdx: i);
    k[0][i].deltaVel = p.getForce() / p.getMass() * dt;
    // k1.v = h * a = h * F/m
    k[0][i].deltaPos = p.getVelocity() * dt;
    // k1.x = h * v
}
```

Set the initial state for compute k2, and recompute all force applied on system. And compute k2 by update params.

k2 = $hf(x_0 + k_1/2, t_0 + h/2)$

```
// k2 = hf(x0 + k1/2, t0 + h/2)
for (int i = 0; i < n; i++) {
    Particle& p = jelly->getParticle(particleIdx: i);
    p.setPosition(_position: x[i] + k[0][i].deltaPos * 0.5f);
    p.setVelocity(_velocity: v[i] + k[0][i].deltaVel * 0.5f);
    p.setForce(_force: Eigen::Vector3f::Zero());
}

if (particleSystem.sceneIdx == 1) {
    particleSystem.computeElevatorForce();
}
particleSystem.computeAllForce();

// k2 = hf(x0 + k1/2, t0 + h/2)
for (int i = 0; i < n; i++) {
    Particle& p = jelly->getParticle(particleIdx: i);
    k[1][i].deltaVel = p.getForce() / p.getMass() * dt;
    k[1][i].deltaPos = p.getVelocity() * dt;
}
```

Compute k3 and k4 by the similar steps.

(k3 = hf(x_0 + k_2/2, t_0 + h/2), k4 = hf(x_0 + k_3, t_0 + h))

(skip the detailed code)

Compute the final result by apply RK4

$x(t_0+h) = x(t_0) + (k1 + 2*k2 + 2*k3 + k4)/6$

And update the particles' position and velocity.

```
//(k1 + 2*k2 + 2*k3 + k4)/6
for (int i = 0; i < n; i++) {
    Particle& p = jelly->getParticle( particleIdx: i);

    Eigen::Vector3f dx =
        (k[0][i].deltaPos + 2.0f * k[1][i].deltaPos + 2.0f * k[2][i].deltaPos + k[3][i].deltaPos) / 6.0f;
    Eigen::Vector3f dv =
        (k[0][i].deltaVel + 2.0f * k[1][i].deltaVel + 2.0f * k[2][i].deltaVel + k[3][i].deltaVel) / 6.0f;

    p.setPosition( _position: x[i] + dx);
    p.setVelocity( _velocity: v[i] + dv);
    p.setForce( _force: Eigen::Vector3f::Zero());
}
```

If it's elevator scene, apply RK4 on updating elevator status.

(Skip the code here, doing the similar steps as we update particles' state)

And set elevator counter to be original elevator counter value, since we will change elevator counter once we use compute elevator force function.

"

```
particleSystem.elevatorCounter = eleCounter;
```

2. Result & Discussion

- **The difference between integrators**

Explicit Euler integration

It's is the simplest method in the integrators and has fastest computational performance. Despite that, it's the least stable of all method. When spring coefficient is high, it requires extremely small time steps to remain stable.

I think jelly looks much softer when using Explicit Euler than using other methods.

Implicit Euler integration

Perform much better than explicit Euler integration in high spring

coefficient, but it's more computationally expensive since it need to solve a system of equations to find the next state.

Midpoint Euler integration

It's more stable than explicit Euler method but less stable than implicit Euler. Besides, it requires two force evaluations per step, making it slower than explicit Euler but still faster than RK4.

Runge-Kutta 4th Order Integration

Has highest stability for non-stiff systems and highest accuracy. But it's most computationally expensive, requiring four force evaluations per step.

The jelly using RK4 method goes down more slowly than simulations using other methods.

- **Effect of parameters**

If we adjust the spring coefficient to lower value, Explicit Euler method seems has some performance problem.

It's the result simulated by explicit Euler method under spring coefficient 1000, one corner of jelly has been incomplete.

Simulations by using other method can get complete jelly, and the simulation by using implicit Euler method make the jelly looks much softer.

Besides, if we adjust damper coefficient higher, the jelly tend to has less rotations before falling down the slope.

3. Bonus

*** The png files of cat is enclosed in the folder texture**

I adjust appearance of jelly. Instead using slime appearance, I upload the photo of front-look and side-look of a cat. Make the jelly has a cartoon cat graph on its interface.

Below is the cat jelly.

4. Conclusion

Based on the evaluation, different integration has distinct pros and cons. Explicit Euler provide least computational time but has poor stability. RK4 offers highest accuracy but greatest computational cost.

Besides, coefficient affect the simulation behavior, lower spring coefficient challenges the stability, while higher damper coefficient reduces jelly rotation before falling.

Simulations by using RK4 seems to be stranger, since jelly falls slower compared to other simulation methods. I think there's still somewhere can be improve to make it perform better.