HW1

111550143 林彥佑

- Implementation

TODO#1 接彈簧

```
for (int i = 0; i < particleNumPerEdge; i++) {
    for (int j = 0; j < particleNumPerEdge; j++) {
        for (int k = 0; k < particleNumPerEdge - 1; k++) {

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

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

照講義的題是先接上 Struct Spring，分別以三個不同方向(x, y, z)處理連接避免重複，x, y, z 方向的連接程式碼不同之處僅在於 iNeighborID，分別用 1, particleNumPerEdge, particleumPerFace 找連接的 particle。

再來 Bending Spring 的接法與 Struct Spring 類似，差別只是 Bending 會跨兩個接，所以在處理時會以 2, 2*particleNumPerEdge, 2*particleumPerFace 找連接的 particle。

```
std::vector<std::vector<int>> offset = {
    {-1, -1, -1}, {-1, -1,  0}, {-1, -1,  1},
    {-1,  0, -1}, {-1,  0,  1},
    {-1,  1, -1}, {-1,  1,  0}, {-1,  1,  1},
    { 0, -1, -1}, { 0, -1,  1},
    { 0,  1, -1}, { 0,  1,  1},
    { 1, -1, -1}, { 1, -1,  0}, { 1, -1,  1},
    { 1,  0, -1}, { 1,  0,  1},
    { 1,  1, -1}, { 1,  1,  0}, { 1,  1,  1}
};
```

```cpp
for (int i = 0; i < particleNumPerEdge; i++) {
    for (int j = 0; j < particleNumPerEdge; j++) {
        for (int k = 0; k < particleNumPerEdge; k++) {
            int iParticleID = i * particleNumPerFace + j * particleNumPerEdge + k;

            for (const auto& dir : offset) {
                int ni = i + dir[0];
                int nj = j + dir[1];
                int nk = k + dir[2];

                if (ni >= 0 && ni < particleNumPerEdge &&
                    nj >= 0 && nj < particleNumPerEdge &&
                    nk >= 0 && nk < particleNumPerEdge &&
                    iParticleID < ni * particleNumPerFace + nj * particleNumPerEdge + nk) {

                    int iNeighborID = ni * particleNumPerFace + nj * particleNumPerEdge + nk;

                    Eigen::Vector3f SpringStartPos = particles[iParticleID].getPosition();
                    Eigen::Vector3f SpringEndPos = particles[iNeighborID].getPosition();
                    Eigen::Vector3f Length = SpringStartPos - SpringEndPos;
                    float absLength = sqrt(Length[0] * Length[0] + Length[1] * Length[1] + Length[2] * Length[2]);

                    springs.push_back(Spring(iParticleID, iNeighborID, absLength, springCoefStruct, damperCoefStruct,
                                             Spring::SpringType::SHEAR));
                    shear_num++;
                }
            }
        }
    }
}
```

在做 Shear Spring 時，我選擇把 20 個不同方向都列出來，並且在處理時增加判斷條件，只有在 iParticleID<iNeighborID 時才做連接，這樣雖然說會增加判斷，但是最直覺確保連接埠重複的方法。

TODO#2 Spring Force & Damper Force & Internal Force

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

    Eigen::Vector3f spring_force = Eigen::Vector3f::Zero();
    Eigen::Vector3f L = positionA - positionB;

    spring_force = -1 * springCoef * (L.norm() - restLength) * L / L.norm();

    return spring_force;
}

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 damper_force = Eigen::Vector3f::Zero();
    Eigen::Vector3f L = positionA - positionB;
    Eigen::Vector3f V = velocityA - velocityB;

    damper_force = -1 * damperCoef * (V.dot(L) / L.norm()) * L / L.norm();

    return damper_force;
}
```

算彈力套用 particle 講義 p10 所提供的公式: $= -k_s(|\vec{x}_a - \vec{x}_b| - r)\frac{\vec{x}_a - \vec{x}_b}{|\vec{x}_a - \vec{x}_b|}$

算 Damper Force 套用講義 p11 所提供的公式: $= -k_d \frac{(\vec{v}_a - \vec{v}_b)\cdot(\vec{x}_a - \vec{x}_b)}{|\vec{x}_a - \vec{x}_b|}\frac{(\vec{x}_a - \vec{x}_b)}{|\vec{x}_a - \vec{x}_b|}$

然後在計算 internal force 時將這兩個力納入:

```cpp
startParticle.addForce(springForce + damperForce);
endParticle.addForce(-1 * springForce - damperForce);
```

TODO#3 處理平面碰撞與 contact force

```cpp
void PlaneTerrain::handleCollision(const float delta_T, Jelly& jelly) {
    Eigen::Vector3f normal = Eigen::Vector3f(sinf(PI/9), cosf(PI/9), 0);
    Eigen::Vector3f PointOfPlane = Eigen::Vector3f(0, 0, 0);
    Eigen::Vector3f n = normal.normalized();

    for (int i = 0; i < jelly.getParticleNum(); i++) {
        Particle& particle = jelly.getParticle(i);
        Eigen::Vector3f x = particle.getPosition();
        Eigen::Vector3f v = particle.getVelocity();
        Eigen::Vector3f v_n = v.dot(n) * n;
        Eigen::Vector3f v_t = v - v_n;
        Eigen::Vector3f force = particle.getForce();

        // collision detection
        if(n.dot(x - PointOfPlane) < eEPSILON && v.dot(n) < 0)
        {
            Eigen::Vector3f v_n_new = -1 * coefResist * v_n;
            Eigen::Vector3f v_t_new = v_t;
            particle.setVelocity(v_n_new + v_t_new);

            // contact force
            if(std::abs(n.dot(x - PointOfPlane)) < eEPSILON || std::abs(v.dot(n)) < eEPSILON)
            {
                if(n.dot(force) < 0){
                    Eigen::Vector3f frictionForce = -1 * coefFriction * (-n.dot(force)) * v_t.normalized();
                    Eigen::Vector3f contactForce = -1 * (n.dot(force)) * n;
                    particle.addForce(frictionForce + contactForce);
                }
            }
        }
    }
}
```

在處理斜坡時，我先將法向量寫出，因為是 XZ 平面並轉-20 度，故法向量為(sinf(PI/9), cosf(PI/9), 0)

，平面上一點我選定(0, 0, 0)，並將速度分成垂直平面以及平行平面，再來判斷 particle 是否與斜坡發生 collision: (particle p15)

$$\mathbf{N}\cdot(\mathbf{x}-\mathbf{p})<\varepsilon \quad \mathbf{N}\cdot\mathbf{v}<0$$

更新碰撞後的速度，因為斜坡沒速度，所以套用 particle p17 之公式

$$\mathbf{v}'=-k_r\,\mathbf{v}_N+\mathbf{v}_T$$

接著處理 contact force 與 friction force，參考 particle p18 提供的 contact conditions:

- On the wall
  $$|\mathbf{N}\cdot(\mathbf{x}-\mathbf{p})|<\varepsilon$$
- Moving along the wall
  $$|\mathbf{N}\cdot\mathbf{v}|<\varepsilon$$

並在 N·f<0 時需計算 contact force 與 friction force，套用 particle p19 提供的公式

$$\mathbf{f}^c=-(\mathbf{N}\cdot\mathbf{f})\,\mathbf{N} \quad \mathbf{f}^f=-k_f(-\mathbf{N}\cdot\mathbf{f})\,v_t$$

```cpp
// collision detection
if(n.dot(x - PointOfPlane) < eEPSILON && (v - elevator_v).dot(n) < 0)
{
    float u_a = v.dot(n);
    float u_b = elevator_v.dot(n);
    Eigen::Vector3f v_t = v - u_a * n;

    float v_a = (particle_m * u_a + elevator_m * u_b + elevator_m * coefResist * (u_b - u_a)) / (particle_m + elevator_m);

    particle.setVelocity(v_t + v_a * n);

    // contact force
    if(std::abs(n.dot(x - PointOfPlane)) < eEPSILON || std::abs(v.dot(n)) < eEPSILON)
    {
        if(n.dot(force) < 0){
            Eigen::Vector3f frictionForce = -1 * coefFriction * (-n.dot(force)) * v_t.normalized();
            Eigen::Vector3f contactForce = -1 * (n.dot(force)) * n;
            particle.addForce(frictionForce + contactForce);
        }
    }
}
```

Elevator 的話要考慮其本身的速度，故用 HW1 提供的碰撞公式，而非 particle 講義中的

$$v_a = \frac{m_a u_a + m_b u_b + m_b e(u_b - u_a)}{m_a + m_b}$$

TODO#4 Integrator

1. ExplicitEuler

```cpp
void ExplicitEulerIntegrator::integrate(MassSpringSystem& particleSystem) {
    // TODO#4-1: Integrate position and velocity
    float dt = particleSystem.deltaTime;

    for (int i = 0; i < particleSystem.getJellyCount(); i++) {
        Jelly* jelly = particleSystem.getJellyPointer(i);
        for (int j = 0; j < jelly->getParticleNum(); j++) {
            Particle& particle = jelly->getParticle(j);
            Eigen::Vector3f acceleration = particle.getAcceleration();
            Eigen::Vector3f velocity = particle.getVelocity();
            Eigen::Vector3f position = particle.getPosition();


            position = position + velocity * dt;
            velocity = velocity + acceleration * dt;

            particle.setPosition(position);
            particle.setVelocity(velocity);
            particle.setForce(Eigen::Vector3f::Zero());
        }
    }

    if (particleSystem.sceneIdx == 1 && particleSystem.elevatorTerrain != nullptr) {
        Terrain& elevator = *(particleSystem.elevatorTerrain);

        Eigen::Vector3f a = elevator.getAcceleration();
        Eigen::Vector3f v = elevator.getVelocity();
        Eigen::Vector3f x = elevator.getPosition();

        Eigen::Vector3f v_new = v + dt * a;
        Eigen::Vector3f x_new = x + dt * v;

        elevator.setVelocity(v_new);
        elevator.setPosition(x_new);
        elevator.setForce(Eigen::Vector3f::Zero());
    }
```

利用 Explicit Euler 處理 jelly 中每個 particle 的速度與位置，公式為

$$\mathbf{x}(t+h) = \mathbf{x}(t) + h \cdot f(\mathbf{x}, t)$$

用當前位置、速度、加速度，更新 delta t 後的位置與速度。

sceneID=1 時表示為 elevator，將 elevator 當作一個 particle，用相同方法更新速度與位置。

2. ImplicitEuler

```cpp
float dt = particleSystem.deltaTime;
int jellyCount = particleSystem.getJellyCount();
int elevatorCounterBU = particleSystem.elevatorCounter;

std::vector<std::vector<Eigen::Vector3f>> oldPos(jellyCount);
std::vector<std::vector<Eigen::Vector3f>> oldVel(jellyCount);
Eigen::Vector3f ElevatorPos = particleSystem.elevatorTerrain->getPosition();
Eigen::Vector3f ElevatorVel = particleSystem.elevatorTerrain->getVelocity();
```

```cpp
for(int i = 0; i < jellyCount; i++) {
    Jelly* jelly = particleSystem.getJellyPointer(i);
    int particleNum = jelly->getParticleNum();
    oldPos[i].resize(particleNum);
    oldVel[i].resize(particleNum);
    for (int j = 0; j < particleNum; j++) {
        Particle& particle = jelly->getParticle(j);
        oldPos[i][j] = particle.getPosition();
        oldVel[i][j] = particle.getVelocity();
        particle.setPosition(particle.getPosition());
        particle.setVelocity(particle.getVelocity());
        particle.setForce(Eigen::Vector3f::Zero());
    }
}

for(int i = 0; i < jellyCount; i++) {
    Jelly* jelly = particleSystem.getJellyPointer(i);
    particleSystem.computeJellyForce(*jelly);
}

for(int i = 0; i < jellyCount; i++) {
    Jelly* jelly = particleSystem.getJellyPointer(i);
    for(int j = 0; j < jelly->getParticleNum(); j++) {
        Particle& particle = jelly->getParticle(j);
        Eigen::Vector3f acceleration = particle.getAcceleration();
        Eigen::Vector3f v_new = oldVel[i][j] + acceleration * dt;
        Eigen::Vector3f x_new = oldPos[i][j] + particle.getVelocity() * dt;

        particle.setPosition(x_new);
        particle.setVelocity(v_new);
        particle.setForce(Eigen::Vector3f::Zero());
    }
}
```

```cpp
if (particleSystem.sceneIdx == 1 && particleSystem.elevatorTerrain != nullptr) {
    Terrain& elevator = *(particleSystem.elevatorTerrain);

    elevator.setVelocity(ElevatorVel);
    elevator.setPosition(ElevatorPos);
    elevator.setForce(Eigen::Vector3f::Zero());

    particleSystem.computeElevatorForce();

    Eigen::Vector3f v_new = ElevatorVel + elevator.getAcceleration() * dt;
    Eigen::Vector3f x_new = ElevatorPos + elevator.getVelocity() * dt;

    particleSystem.elevatorCounter = elevatorCounterBU;

    elevator.setVelocity(v_new);
    elevator.setPosition(x_new);
    elevator.setForce(Eigen::Vector3f::Zero());
}
```

Implicit 是要用現在位置與下個時間點的速度來計算下個時間點之位置，故我先將此時間點之位置與速度存入 oldPos 與 oldVel，並用現在位置與速度去計算 Force，得出下個時間點的速度與加速度，得出在此時的下個時間點之位置與速度，並 clear force。

$$\mathbf{x}_{n+1} = \mathbf{x}_n + hf(\mathbf{x}_{n+1}, t_{n+1})$$

同理，將 elevator 視為一個 particle 同樣進行 implicit Euler，但要記的把 elevatorCounter 更新回去，因為這裡只是為了模擬，並不是要推進 Counter。

3. MidpointEuler

```
for(int i = 0; i < jellyCount; i++) {
    int particleNum = jelly->getParticleNum();
    oldPos[i].resize(particleNum);
    oldVel[i].resize(particleNum);
    for (int j = 0; j < particleNum; j++) {
        Particle& particle = jelly->getParticle(j);
        oldPos[i][j] = particle.getPosition();
        oldVel[i][j] = particle.getVelocity();
        Eigen::Vector3f acceleration = particle.getAcceleration();
        Eigen::Vector3f v_mid = particle.getVelocity() + acceleration * dt / 2.0f;
        Eigen::Vector3f x_mid = particle.getPosition() + particle.getVelocity() * dt / 2.0f;

        particle.setPosition(x_mid);
        particle.setVelocity(v_mid);
        particle.setForce(Eigen::Vector3f::Zero());
    }
}

for(int i = 0; i < jellyCount; i++) {
    Jelly* jelly = particleSystem.getJellyPointer(i);
    particleSystem.computeJellyForce(*jelly);
}

for(int i = 0; i < jellyCount; i++) {
    Jelly* jelly = particleSystem.getJellyPointer(i);
    for(int j = 0; j < jelly->getParticleNum(); j++) {
        Particle& particle = jelly->getParticle(j);
        Eigen::Vector3f acceleration = particle.getAcceleration();
        Eigen::Vector3f v_new = oldVel[i][j] + acceleration * dt;
        Eigen::Vector3f x_new = oldPos[i][j] + particle.getVelocity() * dt;

        particle.setPosition(x_new);
        particle.setVelocity(v_new);
        particle.setForce(Eigen::Vector3f::Zero());
    }
```

與 implicit 類似，但這次要用現在的位置與預測的下一個位置的中點之速度去預估下一個點，公式

$$\mathbf{x}(t_0 + h) = \mathbf{x}(t_0) + h \cdot f_{mid}$$

所以存完現在位置與速度後，設定位置是中點位置與速度，不再是現在位置與速度，然後計算 force 得新的速度與加速度與舊位置與速度得出下一個預估之位置與速度，最後 clear force。

同樣將 elevator 計算 midpoint Euler，如 implicit 的方法。

4. RungeKutta

```
std::vector<std::vector<StateStep>> k1(jellyCount);
std::vector<std::vector<StateStep>> k2(jellyCount);
std::vector<std::vector<StateStep>> k3(jellyCount);
std::vector<std::vector<StateStep>> k4(jellyCount);
StateStep k1_elevator, k2_elevator, k3_elevator, k4_elevator;

for(int i = 0; i < jellyCount; i++) {
    Jelly* jelly = particleSystem.getJellyPointer(i);
    int particleNum = jelly->getParticleNum();
    oldPos[i].resize(particleNum);
    oldVel[i].resize(particleNum);
    k1[i].resize(particleNum);
    k2[i].resize(particleNum);
    k3[i].resize(particleNum);
    k4[i].resize(particleNum);
    for (int j = 0; j < particleNum; j++) {
        Particle& particle = jelly->getParticle(j);
        oldPos[i][j] = particle.getPosition();
        oldVel[i][j] = particle.getVelocity();
    }
}
```

```
for(int i = 0; i < jellyCount; i++) {
    Jelly* jelly = particleSystem.getJellyPointer(i);
    for (int j = 0; j < jelly->getParticleNum(); j++) {
        Particle& particle = jelly->getParticle(j);
        Eigen::Vector3f acceleration = particle.getAcceleration();
        Eigen::Vector3f velocity = particle.getVelocity();
        Eigen::Vector3f position = particle.getPosition();

        k1[i][j].deltaVel = acceleration * dt;
        k1[i][j].deltaPos = velocity * dt;

        particle.setPosition(oldPos[i][j] + k1[i][j].deltaPos * 0.5f);
        particle.setVelocity(oldVel[i][j] + k1[i][j].deltaVel * 0.5f);
        particle.setForce(Eigen::Vector3f::Zero());
    }
}

for(int i = 0; i < jellyCount; i++) {
    Jelly* jelly = particleSystem.getJellyPointer(i);
    particleSystem.computeJellyForce(*jelly);
}
```

```
for(int i = 0; i < jellyCount; i++) {
    Jelly* jelly = particleSystem.getJellyPointer(i);
    for (int j = 0; j < jelly->getParticleNum(); j++) {
        Particle& particle = jelly->getParticle(j);
        Eigen::Vector3f acceleration = particle.getAcceleration();
        Eigen::Vector3f velocity = particle.getVelocity();
        Eigen::Vector3f position = particle.getPosition();

        k2[i][j].deltaVel = acceleration * dt;
        k2[i][j].deltaPos = velocity * dt;

        particle.setPosition(oldPos[i][j] + k2[i][j].deltaPos * 0.5f);
        particle.setVelocity(oldVel[i][j] + k2[i][j].deltaVel * 0.5f);
        particle.setForce(Eigen::Vector3f::Zero());
    }
}

for(int i = 0; i < jellyCount; i++) {
    Jelly* jelly = particleSystem.getJellyPointer(i);
    particleSystem.computeJellyForce(*jelly);
}
```

```
for(int i = 0; i < jellyCount; i++) {
    for (int j = 0; j < jelly->getParticleNum(); j++) {
        Particle& particle = jelly->getParticle(j);
        Eigen::Vector3f acceleration = particle.getAcceleration();
        Eigen::Vector3f velocity = particle.getVelocity();
        Eigen::Vector3f position = particle.getPosition();

        k3[i][j].deltaVel = acceleration * dt;
        k3[i][j].deltaPos = velocity * dt;

        particle.setPosition(oldPos[i][j] + k3[i][j].deltaPos);
        particle.setVelocity(oldVel[i][j] + k3[i][j].deltaVel);
        particle.setForce(Eigen::Vector3f::Zero());
    }
}

for(int i = 0; i < jellyCount; i++) {
    Jelly* jelly = particleSystem.getJellyPointer(i);
    particleSystem.computeJellyForce(*jelly);
}

for(int i = 0; i < jellyCount; i++) {
    Jelly* jelly = particleSystem.getJellyPointer(i);
    for (int j = 0; j < jelly->getParticleNum(); j++) {
        Particle& particle = jelly->getParticle(j);
        Eigen::Vector3f acceleration = particle.getAcceleration();
        Eigen::Vector3f velocity = particle.getVelocity();
        Eigen::Vector3f position = particle.getPosition();

        k4[i][j].deltaVel = acceleration * dt;
        k4[i][j].deltaPos = velocity * dt;
    }
}
```

```
for(int i = 0; i < jellyCount; i++) {
    Jelly* jelly = particleSystem.getJellyPointer(i);
    for (int j = 0; j < jelly->getParticleNum(); j++) {
        Particle& particle = jelly->getParticle(j);

        Eigen::Vector3f x_new = oldPos[i][j] + (k1[i][j].deltaPos + 2 * k2[i][j].deltaPos + 2 * k3[i][j].deltaPos + k4[i][j].deltaPos) / 6.0f;
        Eigen::Vector3f v_new = oldVel[i][j] + (k1[i][j].deltaVel + 2 * k2[i][j].deltaVel + 2 * k3[i][j].deltaVel + k4[i][j].deltaVel) / 6.0f;

        particle.setPosition(x_new);
        particle.setVelocity(v_new);
        particle.setForce(Eigen::Vector3f::Zero());
    }
}
```

Runge-Kutta 4 使用多階段預測。首先備份原始位置與速度，並依序計算四組微小變化量 k1 ~ k4，其中每組皆透過暫時更新 particle 狀態並重新計算速度與加速度來取得。最後將這四組權重平均後，計算出下一時間點的位置與速度。

同理，Elevator 只是另一個 particle，用同樣方法計算出速度與位置，並 reset counter。

參考公式:

$$\mathbf{x}(t_0+h)=\mathbf{x}(t_0)+\frac{1}{6}(k_1+2k_2+2k_3+k_4)+O(h^5)$$

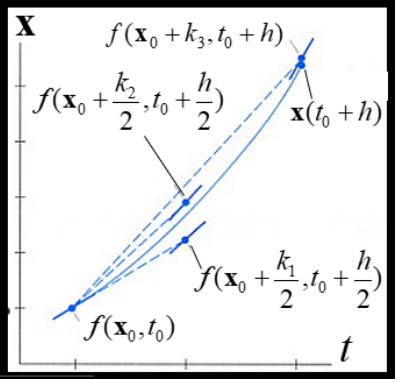
- Using a weighted average of slopes obtained at four points

$$k_1 = hf(\mathbf{x}_0, t_0)$$

$$k_2 = hf(\mathbf{x}_0+\frac{k_1}{2}, t_0+\frac{h}{2})$$

$$k_3 = hf(\mathbf{x}_0+\frac{k_2}{2}, t_0+\frac{h}{2})$$

$$k_4 = hf(\mathbf{x}_0+k_3, t_0+h)$$



- Result & Discussion

在變數皆不改變，四種方法經實測後僅有些微差異，implicit Euler 在斜坡滾動較少，較早開始滑行，應該為他採用速度為下一個預測之時間點之速度導致，Runge-Kutta 因為計算量較大，無論斜坡還是升降梯顯示效果相較其他方法像是慢動作執行，但也最為自然。

然後進行不同變數修改的測試(springCoef, damperCoef, coefResist, coefFriction)

1. springCoef

當 springCoef 越大，Jelly 的彈性力越強，變形時產生的回復力越大，會有更強的反彈力，如 implicit 在把 springCoef 從 1400 調到 1500 後，他在斜坡上因彈跳多一次翻滾，表示彈力增加，更極端的例子，將 springCoef 調低至 100，則發現 jelly 下半部被壓扁看不出回復的跡象，第一次碰觸地面也沒有反彈之跡象，調高至 2800 發現，在彈力上的視覺效果好很多，剛度更好。

2. damperCoef

damperCoef 越大，彈簧的振動就會被抑制越快，Jelly 看起來會越黏、重，反彈與彈性效果減弱。

我將 damperCoef 調低至 10，jelly 彈跳後的滯空時間明顯比 50 的多，彈跳效果較好，調高至 80 後，在 elevator 場景時第一次接觸平面時幾乎沒有彈跳，故與理論相符。

3. coefResist

在 handleCollision 中修改 coefResist，並用斜坡場景測試，當 coefResist 為 1 時為完全彈性碰撞，垂直斜面的速度經碰撞後不改變，彈跳比 0.5 好，當等於 0 時，幾乎沒有彈跳只會翻滾，表示只剩下平行斜面的速度。

4. coefFriction

將他調到 10.0，在斜坡場景時幾乎不發生彈跳只翻滾，且無滑行，調至 0 後，明顯增加滑行距離與時間。

但 3、4 在 elevator 上均無明顯差別。

最後在穩定測試，當 springCoef 為 50000，然後 delta t 為 0.0012，explicit, implicit 都出現錯誤無法執行，midpoint 與 Runge-Kutta 活了下來，更進一步把 springCoef 調高至 100000，delta t 為 0.0014 時，midpoint 出現不合理的抖動與緊縮，Runge-Kutta 仍穩定。

- Conclusion

在本次作業中，我實作了 Explicit Euler、Implicit Euler、Midpoint Euler 與 Runge-Kutta 4，並於斜坡與 elevator 場景中測試其行為差異，進一步調整模擬參數，如 springCoef、damperCoef、coefResist、coefFriction，以觀察系統對這些參數的反應。

模擬結果顯示， Explicit Euler、Implicit Euler 雖計算簡單、效率高，但在高剛性參數下極不穩定，容易導致系統崩潰。Midpoint Euler 在穩定性與物理真實性之間取得良好平衡，惟在極端參數條件下仍會出現形變與抖動。Runge-Kutta 4 雖然計算量較大、動畫呈現稍慢，但整體效果最為平滑且符合真實物理行為，即使在極端設定下仍能穩定運作。

在物理性質方面，提升 springCoef 能有效增加彈性與回復力，而增加 damperCoef 則可快速抑制彈跳與振動。碰撞參數方面，coefResist 越大會帶來更強的反彈效果，coefFriction 越小則會顯著增加滑動距離。不過，在電梯場景中這兩個參數的影響較不明顯，推測可能因為重力與垂直運動為主而掩蓋其作用。

總結而言，模擬行為對於積分器選擇、delta t 以及物理參數的組合非常敏感。本作業使我更深入理解數值方法與物理建模在電腦動畫與特效的關係與重要性。