電腦動畫與特效 HW1 110612117 張仲瑜

I. Introduction/Motivation

This project is designed to make us be familiar with the dynamics particle system. Simulating the cloth with springs consisting of particles, detecting and reacting collision between particles and sphere, and using different integrators to simulate the position, velocity and acceleration are included in the project.

II. Fundamental

A. Internal Force

Utilize the spring–damper model to simulate the internal forces within the cloth. Each particle is connected to its neighbors by springs, which provides the restoring force to simulate the elasticity of cloth. The dampers mitigate oscillation to simulate the dissipation of energy in real world.

B. Collision and restoration of penetration

Formula of collision model with restitution coefficient is used to present the collision between the cloth and the sphere. The velocities are updated after the calculation.

$$v_a' = \frac{m_a u_a + m_b u_b + m_b C_R(u_b - u_a)}{m_a + m_b}$$
$$v_b' = \frac{m_a u_a + m_b u_b + m_a C_R(u_a - u_b)}{m_a + m_b}$$

Restoration of penetration happens when the distance between particles and sphere is smaller than the radius of the sphere. The position of particles which penetrate the sphere will be countervailed to the surface.

C. Integrators

Use different ways to approximate the position and velocity of particles.

i. Explicit Euler: Use delta time and current derivative to approximate.

ii. Implicit Euler: Use the derivative of next step instead of current derivative to approximate.

iii. Midpoint Euler: Use the Explicit Euler to get the derivative state after 1/2 step, and approximate with the Explicit Euler again.

iv. Runge–Kutta Fourth: Different derivatives of 4 states are calculate with Explicit Euler and mixed them in proportion, and then approximate with the Explicit Euler again.

III. Implementation

A. Springs initialization

```
float structrualLength = (_particles.position(0) - _particles.position(1)).norm();
for (int i = 0; i < particlesPerEdge; ++i) {
  for (int j = 0; j < particlesPerEdge - 1; ++j) {
    int index = i * particlesPerEdge + j;
    _springs.emplace_back(index, index + 1, structrualLength, Spring::Type::STRUCTURAL);
  }
}

for (int i = 0; i < particlesPerEdge - 1; ++i) {
  for (int j = 0; j < particlesPerEdge; ++j) {
    int index = i * particlesPerEdge + j;
    _springs.emplace_back(index, index + particlesPerEdge, structrualLength, Spring::Type::STRUCTURAL);
  }
}
```

Initialize the horizontal structural spring by connecting index and index + 1 particles, while initiating the vertical structural spring by connecting index and index + particlePerEdge particles.

```
float bendingLength = 2 * structrualLength;
for (int i = 0; i < particlesPerEdge; ++i) {
  for (int j = 0; j < particlesPerEdge - 2; ++j) {
    int index = i * particlesPerEdge + j;
    _springs.emplace_back(index, index + 2, bendingLength, Spring::Type::BEND);
  }
}

for (int i = 0; i < particlesPerEdge - 2; ++i) {
  for (int j = 0; j < particlesPerEdge; ++j) {
    int index = i * particlesPerEdge + j;
    _springs.emplace_back(index, index + 2 * particlesPerEdge, bendingLength, Spring::Type::BEND);
  }
}
```

Similar way can be used to initialize the bending springs. They are twice as long as structural springs.

```
float shearingLength = sqrt(2) * structrualLength;
for (int i = 0; i < particlesPerEdge - 1; ++i) {
 for (int j = 0; j < particlesPerEdge - 1; ++j) {
  int index = i * particlesPerEdge + j;
  _springs.emplace_back(index, index + particlesPerEdge + 1, shearingLength, Spring::Type::SHEAR);
  _springs.emplace_back(index + 1, index + particlesPerEdge, shearingLength, Spring::Type::SHEAR);
 }
}
```

Build the “X” shaped shearing springs by connecting the particles of left–up, right–down and right–up, left–down. They are sqrt(2) times as long as structural springs.

B. Spring force computation

```
for (auto& spring : _springs) {
    int startIdx = spring.startParticleIndex();
    int endIdx = spring.endParticleIndex();
    Eigen::Vector4f startPos = _particles.position(startIdx);
    Eigen::Vector4f endPos = _particles.position(endIdx);
    Eigen::Vector4f springVector = endPos - startPos;
    float currentLength = springVector.norm();
    float springForceMagnitude = springCoef * (currentLength - spring.length());
    Eigen::Vector4f springForceDirection = springVector.normalized();
    Eigen::Vector4f springForce = springForceMagnitude * springForceDirection;

    Eigen::Vector4f velocityDiff = _particles.velocity(endIdx) - _particles.velocity(startIdx);
    float damperForceMagnitude = damperCoef * velocityDiff.dot(springForceDirection);
    Eigen::Vector4f damperForce = damperForceMagnitude * springForceDirection;
    Eigen::Vector4f totalForce = springForce + damperForce;

    _particles.acceleration(startIdx) += totalForce * _particles.inverseMass(startIdx);
    _particles.acceleration(endIdx) -= totalForce * _particles.inverseMass(endIdx);
}
```

For spring force, calculate the magnitude and the direction to get the force exerted on the particle. For damper force, the calculation of velocity difference, magnitude and the direction are needed.

Noted that the sign applied to the end index should ne

negative due to the restoration property.

C. Collide detection and penetration restoration

```
for (int s = 0; s < sphereCount; s++) {

    Eigen::Vector4f sphereVel = this->_particles.velocity(s);
    Eigen::Vector4f spherePos = this->_particles.position(s);
    float sphereMass = _particles.mass(s);
    float sphereRadius = _radius[s];
    for (int p = 0; p < particlesPerEdge * particlesPerEdge; p++) {
        Eigen::Vector4f particlePos = cloth->particles().position(p);
        Eigen::Vector4f particleVel = cloth->particles().velocity(p);
        float particleMass = cloth->particles().mass(p);

        Eigen::Vector3f delta = (spherePos - particlePos).head<3>();
        float distance = delta.norm();

        Eigen::Vector3f collisionNormal = delta.normalized();
        if (distance - sphereRadius < eth) {
            Eigen::Vector3f va = particleVel.head<3>();
            Eigen::Vector3f vb = sphereVel.head<3>();
            Eigen::Vector3f va_prime =
            (va * particleMass + sphereMass * vb + coefRestitution * sphereMass * (vb - va)) /
            (particleMass + sphereMass);
            Eigen::Vector3f vb_prime =
            (vb * sphereMass + particleMass * va + coefRestitution * particleMass * (va - vb)) /
            (particleMass + sphereMass);
            cloth->particles().velocity(p).head<3>() = va_prime;
            this->_particles.velocity(s).head<3>() = vb_prime;
            float penetrationDepth = sphereRadius + eth - distance;
            Eigen::Vector3f correction = penetrationDepth * collisionNormal;
            cloth->particles().position(p).head<3>() -= correction;
        }
```

Use the formula mentioned above to calculate the velocity and the position after collision.

When distance − sphere radius < eth, which means penetration has happened, the correction of depth * normal should add back to the particle.

It's negative, so change sign to fix it.

D. Integrator

i. Explicit Euler

```
float dt = deltaTime;
for (auto &p : particles) {
    p->velocity() += p->acceleration() * dt;
    p->position() += p->velocity() * dt;
}
```

Iterate all the particles and add the velocity and position with acceleration * delta time and velocity * delta time, which are the derivative of them.

ii. Implicit Euler

```
    std::vector<Eigen::Matrix4Xf> originalPositions(particles.size()), originalVelocities(particles.size());

    for (size_t i = 0; i < particles.size(); ++i) {
        originalPositions[i] = particles[i]->position();
        originalVelocities[i] = particles[i]->velocity();

        particles[i]->position() += particles[i]->velocity() * deltaTime;
        particles[i]->velocity() += particles[i]->acceleration() * deltaTime;
    }
    simulateOneStep();
    for (size_t i = 0; i < particles.size(); ++i) {
        particles[i]->position() = originalPositions[i] + particles[i]->velocity() * deltaTime;
        particles[i]->velocity() = originalVelocities[i] + particles[i]->acceleration() * deltaTime;
    }
}
```

Use two vectors to backup the original position and velocity of particles, and simulate one step with Explicit Euler to get the derivative at next state. Use that derivative to approximate the result with original data and Explicit Euler.

iii. Midpoint Euler

```
std::vector<Eigen::Matrix4Xf> originalPositions(particles.size()), originalVelocities(particles.size());

  for (size_t i = 0; i < particles.size(); ++i) {
  originalPositions[i] = particles[i]->position();
  originalVelocities[i] = particles[i]->velocity();

  particles[i]->position() += particles[i]->velocity() * deltaTime / 2;
  particles[i]->velocity() += particles[i]->acceleration() * deltaTime / 2;
  }
  simulateOneStep();
  for (size_t i = 0; i < particles.size(); ++i) {
  particles[i]->position() = originalPositions[i] + particles[i]->velocity() * deltaTime;
  particles[i]->velocity() = originalVelocities[i] + particles[i]->acceleration() * deltaTime;
  }
}
```

Similar to the last one, but here we use the derivative after 1/2 step, and the rest is the same.

iv. Runge–Kutta Fourth

```
  std::vector<Eigen::Matrix4Xf>
      originalPositions(particles.size()), originalVelocities(particles.size()),
      k1delPositions(particles.size()), k1delVelocities(particles.size()),
      k2delPositions(particles.size()), k2delVelocities(particles.size()),
      k3delPositions(particles.size()), k3delVelocities(particles.size()),
      k4delPositions(particles.size()), k4delVelocities(particles.size());
  for (size_t i = 0; i < particles.size(); ++i) {
      originalPositions[i] = particles[i]->position();
      originalVelocities[i] = particles[i]->velocity();

      k1delPositions[i] = particles[i]->velocity() * deltaTime;
      k1delVelocities[i] = particles[i]->acceleration() * deltaTime;
      particles[i]->position() += k1delPositions[i] / 2;
      particles[i]->velocity() += k1delVelocities[i] / 2;
  }
  simulateOneStep();
  for (size_t i = 0; i < particles.size(); ++i) {
      k2delPositions[i] = particles[i]->velocity() * deltaTime;
      k2delVelocities[i] = particles[i]->acceleration() * deltaTime;
      particles[i]->position() = originalPositions[i] + k2delPositions[i] / 2;
      particles[i]->velocity() = originalVelocities[i] + k2delVelocities[i] / 2;
  }
  simulateOneStep();
  for (size_t i = 0; i < particles.size(); ++i) {
      k3delPositions[i] = particles[i]->velocity() * deltaTime;
      k3delVelocities[i] = particles[i]->acceleration() * deltaTime;
      particles[i]->position() = originalPositions[i] + k3delPositions[i];
      particles[i]->velocity() = originalVelocities[i] + k3delVelocities[i];
  }
  simulateOneStep();
  for (size_t i = 0; i < particles.size(); ++i) {
      k4delPositions[i] = particles[i]->velocity() * deltaTime;
      k4delVelocities[i] = particles[i]->acceleration() * deltaTime;
      particles[i]->position() = originalPositions[i] + ((k1delPositions[i] + k2delPositions[i] * 2 + k3delPositions[i] * 2 + k4delPositions[i] * 1) / 6);
      particles[i]->velocity() = originalVelocities[i] + ((k1delVelocities[i] + k2delVelocities[i] * 2 + k3delVelocities[i] * 2 + k4delVelocities[i] * 1) / 6);
  }
}
```

Leverage the idea of mid-point Euler,

Runge-Kutta Fourth use the weighted sum of different steps (also by Explicit Euler) to serve as the derivative, and then use it to do the Explicit Euler with the original data to approximate the result.

The proportion: 1/6 k1, 1/3 k2, 1/3 k3, 1/6 k4

IV. Result and discussion

A. Integrator

	Explicit	Implicit	Midpoint	Runge-Kunta
Accuracy	least	Second last	high	Highest
Computation Cost (FrameRate)	Low 144Hz	Medium 70~72Hz	Medium 68~72Hz	High 30~48Hz

Integrators consider more steps and need root solving from left to right to lower the approximation error, so the computation cost raise.

B. Effect of parameters

i. Spring Coefficient

Higher value leads to a cloth that resists deformation more strongly, and less prone to large displacement, while lower value leads to a softer

one that stretches more easily.

ii. Damping Coefficient

Higher value leads to more energy loss and less responsive, and extremely high value leads to non-responsive cloth. In contrast, lower value results in less energy loss, and extremely low value results in unrealistic behavior which oscillates all the time.

iii. Delta Time

Smaller delta time provides a more refined simulation, while it also raises the computation cost. In contrast, larger delta time provides a speedy simulation, but the error is larger.

V. Bonus

A. Flag -- 0 degree of freedom along the left-hand side edge, with key f to (de)activate it.

```
if (flagActivate) {
    for (int i = particlesPerEdge - 1; i < particlesPerEdge * particlesPerEdge; i += particlesPerEdge) {
        cloth.particles().mass(i) = 0.0f;
    }
}
else {
    for (int i = particlesPerEdge - 1; i < particlesPerEdge * particlesPerEdge; i += particlesPerEdge) {
        cloth.particles().mass(i) = particleMass;
    }
}
```

I add both of them after the 4 pins fix part in main.cpp

Since 0 degree of freedom is applied to the left-hand side edge, we can simply set the mass of those

particles to zero to make their velocity and acceleration to zero, which leads them to fixed particles.

B. Curtain -- 1 degree of freedom along the top side edge, the x-axis is the free axis. Use key c to (de)activate it.

```
if (curtainActivate) {
    for (int i = particlesPerEdge * (particlesPerEdge - 1); i < particlesPerEdge * particlesPerEdge; i++) {
    cloth.particles().velocity(i)[0] = 0.0;
     cloth.particles().velocity(i)[1] = 0.0;
     cloth.particles().velocity(i)[2] = 0.0;
     cloth.particles().acceleration(i)[0] = 0.0;
     cloth.particles().acceleration(i)[1] = 0.0;
     cloth.particles().acceleration(i)[2] = 0.0;
    }
}
```

Set the velocity and acceleration in y axis and z axis direction to zero to cancel the movement in those direction, which leads to 1 degree of freedom movement (x-axis)

C. Simplify the collision model

```
Eigen::Vector4f spherePos = _particles.position(s);
float sphereRadius = _radius[s];

for (int p = 0; p < particlesPerEdge * particlesPerEdge; p++) {
    Eigen::Vector4f particlePos = cloth->particles().position(p);
    Eigen::Vector3f delta = (spherePos - particlePos).head<3>();
    float distance = delta.norm();
    if (distance - sphereRadius < eth) {
        Eigen::Vector3f collisionNormal = delta.normalized();
        Eigen::Vector3f particleVelocity = cloth->particles().velocity(p).head<3>();
        Eigen::Vector3f newVelocity =
            particleVelocity - (1 + coefRestitution) * particleVelocity.dot(collisionNormal) * collisionNormal;
        cloth->particles().velocity(p).head<3>() = newVelocity;
        float penetrationDepth = sphereRadius + eth - distance;
        Eigen::Vector3f correction = penetrationDepth * collisionNormal;
         cloth->particles().position(p).head<3>() -= correction;
    }
}
```

I found that it's more realistic (personal opinion) if we take the mass difference between the particle and the sphere to infinite large, which can further simplify the formula to above, especially when the constraints are applied.

VI. Conclusion

The project made me gain a deeper understanding of how to implement part of the dynamic particle system and utilize different integrator to approximate the state of particle in the future. Although I've encountered countless obstacles in the process, such as using wrong type, miscalling parameter, etc., It's still a precious experience to deep dive into it.