HW1

110550085 房天越

1. Introduction

In this assignment, we need to implement the spring force system, and we need to handle the collision between the cloth and the sphere.

Also, we need to implement four kinds of integrators to integrate the velocity and acceleration.

2. Fundamentals

I. Spring System

$$\mathbf{f}_a = -k_s(|\mathbf{x}_a - \mathbf{x}_b| - l_0)\frac{\mathbf{x}_a - \mathbf{x}_b}{|\mathbf{x}_a - \mathbf{x}_b|} - k_d\left(\frac{(\mathbf{v}_a - \mathbf{v}_b)\cdot(\mathbf{x}_a - \mathbf{x}_b)}{|\mathbf{x}_a - \mathbf{x}_b|}\right)\frac{\mathbf{x}_a - \mathbf{x}_b}{|\mathbf{x}_a - \mathbf{x}_b|}$$

There are three kinds of springs, including struct, shear, and bending, we use the connection between particles to simulate the spring system.

There are also two kinds of forces in springs, which are spring force and damping force, the spring force can be evaluated by Hooke’ s Law, and the damping force is some forces that make

negative work to the system, including forces like resistance forces and friction forces, etc.

II. Collision

In the implementation of collision, there are some key points to implement, which are collision detection, collision handling, and the avoidance of penetration.

All the collisions can be handled by viewing as collision between mass points, and apply the following formula:

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

and

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

C_R is the coefficient of restitution

III. Integrators

In this assignment, we implement four kinds of integrators:

I. Explicit Euler Method

Update based on current situation.

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

II. Implicit Euler Method

Update based on the information we predict at the next

time slot.

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

III. Midpoint Method

Update based on the information at the midpoint.

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

IV. Runge Kutta Fourth Method

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

Update based on the calculated four points; the four points are calculated by:

$$k_1 = hf(\mathbf{x}_0, t_0)$$
$$k_2 = hf(\mathbf{x}_0 + \frac{k_1}{2}, t_0 + \frac{h}{2})$$
$$k_3 = hf(\mathbf{x}_0 + \frac{k_2}{2}, t_0 + \frac{h}{2})$$
$$k_4 = hf(\mathbf{x}_0 + k_3, t_0 + h)$$

3. Implementation

I. Initialize Springs

Connect the spring in two directions, and in the three given kinds, which are struct, shear, and bending, take one direction of them for example here:

```
float structuralLength = (_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, structuralLength, Spring::Type::STRUCTURAL);
    }
}
```

```
// Connect the shear springs.
float shearLength = (_particles.position(0) - _particles.position(particlesPerEdge+1)).norm();
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, shearLength, Spring::Type::SHEAR);
    }
}
```

```
// Connect the bend springs.
float bendLength = (_particles.position(0) - _particles.position(2)).norm();
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, bendLength, Spring::Type::BEND);
    }
}
```

II. Compute Spring Force

Compute the distance between mass points, and the force vector, then apply Hooke’ s Law to compute the spring force. Finally, store it by changing it to the form of acceleration.

```
for (const auto& s : _springs) {
    int s_id = s.startParticleIndex();
    int e_id = s.endParticleIndex();
    auto d = _particles.position(s_id) - _particles.position(e_id);
    auto dnorm = d.norm();
    auto vf = (_particles.position(s_id) - _particles.position(e_id)).normalized();
    auto vv = _particles.velocity(s_id) - _particles.velocity(e_id);
    auto v_l = (vv.dot(d)) / dnorm;
    auto spring_force = -springCoef * (dnorm - s.length()) * vf - damperCoef * v_l * vf;
    //force update
    _particles.acceleration(s_id) += spring_force * _particles.inverseMass(s_id);
    _particles.acceleration(e_id) -= spring_force * _particles.inverseMass(e_id);
}
```

III. Collision

Here we implement the collision between the cloth and the ball. Since we need to avoid penetration, so we detect penetration by

increasing a little bit length from the radius:

```
if (dnorm < _radius[k] +0.01f) {
```

Then, we handle the collision by updating the velocity and position of the cloth particles:

```
auto new_v = (ball_m * ball_v.dot(dist) / dist.dot(dist) * dist + cloth_m * cloth_v.dot(dist) / dist.dot(dist) * dist
    + coefRestitution * (cloth_v.dot(dist) / dist.dot(dist) * dist
    - ball_v.dot(dist) / dist.dot(dist) * dist)) / (ball_m + cloth_m);

auto v_t = cloth_v - cloth_v.dot(dist) / dist.dot(dist) * dist;
pcloth.velocity(i * particlesPerEdge + j) = new_v + v_t;
pcloth.position(i * particlesPerEdge + j) = spherePos +(_radius[k]+0.01f) * dist.normalized();
```

IV. Integrators

I. Explicit Euler Method

Evaluate with v = v0+at and x = x0 + vt.

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

II. Implicit Euler Method

Evaluate by getting the information in the next time slot with simulateOneStep().

```
for (const auto &p : particles) {
    auto curPos = p->position();
    auto curV = p->velocity();
    auto cura = p->acceleration();

    //Compute Xn+1
    p->velocity() += cura * deltaTime;
    p->position() += curV * deltaTime;
    simulateOneStep();
    p->velocity() = curV + cura * deltaTime;
    p->position() = curPos + p->velocity() * deltaTime;

}
```

III. Midpoint Method

Change the deltaTime in implicit method to only half.

```
for (const auto &p : particles) {

    auto curPos = p->position();
    auto curV = p->velocity();
    auto cura = p->acceleration();

    //Compute Xn+1
    p->velocity() += cura * deltaTime / 2;
    p->position() += curV * deltaTime / 2;
    simulateOneStep();
    p->velocity() = curV + cura * deltaTime;
    p->position() = curPos + p->velocity() * deltaTime;
}
```

IV. Runge Kutta Fourth Method

We evaluate the estimated velocity and position for four points, k1, k2, k3, and k4, and finally use the formula we mentioned above to evaluate the new position and velocity.

```
for (const auto &p : particles) {
    auto curPos = p->position();
    auto curV = p->velocity();
    auto cura = p->acceleration();
    //Compute K1
    auto k1v = cura*deltaTime;
    auto k1Pos = curV*deltaTime;
    //Compute K2
    p->velocity() += cura * deltaTime / 2;
    p->position() += k1Pos / 2;
    simulateOneStep();
    auto k2v = curV + p->acceleration() * deltaTime;
    auto k2Pos = k2v * deltaTime;
    p->velocity() = curV;
    p->position() = curPos;
    //Compute K3
    p->velocity() += p->acceleration() * deltaTime / 2;
    p->position() += k2Pos / 2;
    simulateOneStep();
    auto k3v = curV + p->acceleration() * deltaTime;
    auto k3Pos = k3v * deltaTime;
    p->velocity() = curV;
    p->position() = curPos;
    //Compute K4
    p->velocity() += p->acceleration() * deltaTime;
    p->position() += k3Pos;
    simulateOneStep();
    auto k4v = curV + p->acceleration() * deltaTime;
    auto k4Pos = k4v * deltaTime;
    //Compute Xn+1
    auto newPos = curPos + (k1Pos + 2 * k2Pos + 2 * k3Pos + k4Pos) / 6;
    p->position() = newPos;
    p->velocity() = curV + p->acceleration() * deltaTime;
}
```

4. Result and Discussion

I. The Difference between Integrators

1. Explicit Euler Method

   This is a first-order integrator and the simplest method, because it doesn’ t require too much computation resources, so it can have the highest fps among the four methods.

   However, it is not as stable and accurate compared to other methods, when the coefficient is too high, it fails to simulate.

2. Implicit Euler Method

   This method is also a first-order integrator, but requires the information on the next time slot to evaluate the value in the next time slot. This method is slower than the explicit one, but is more stable and accurate.

3. Midpoint Method

   The method is a second-order integrator, and is very similar with the implicit Euler method, this method provides a similar result with the implicit method, but is slightly more stable.

4. Runge-Kutta Fourth Method

   This method is a fourth-order integrator. Since we use the

weighted average of four points during the evaluation, this has the most stable and accurate result compared to the other methods. However, it also has the worst fps due to its high computation resources requirements.

II. Effect of parameters

1. Spring Coefficient

The spring coefficient affects the stiffness of the spring, if it is larger, then the spring would be more difficult to be pulled, and would also be easier to restore, and vice versa.

2. Damper Coefficient

The damper coefficient affects the damper force of the system, if the damper coefficient is larger, then the system would resist the oscillation more, and vice versa.

5. Conclusion

In this assignment, we learned how to simulation the particle system by simulating the cloth and collision between the cloth and the ball. Also, we learned to use the four integrators to evaluate the velocity and position.

With this assignment, I learned how to apply the theories by

implementation, and visualize the theories, so I learned a lot from it!