# HW1

111550088 張育維

- Introduction/Motivation

  This homework is a simulation of dropping cloth texture on a sphere. We can change some constants in the panel, such as deltaTime, dampercoef, etc.

- Fundamentals

  Spring force:

$$\vec{f}_a = -k_s(|\vec{x}_a - \vec{x}_b| - r)\vec{l}, \quad k_s > 0$$
$$= -k_s(|\vec{x}_a - \vec{x}_b| - r)\frac{\vec{x}_a - \vec{x}_b}{|\vec{x}_a - \vec{x}_b|}$$

  This calculation involves the difference in variation of length between two particles and the original length, multiplied by a coefficient. This result is then used to scale the position vector.

  Damper force:

$$\vec{f}_a = -k_d((\vec{v}_a - \vec{v}_b)\cdot\vec{l})\vec{l}, \quad k_d > 0$$
$$= -k_d \frac{(\vec{v}_a - \vec{v}_b)\cdot(\vec{x}_a - \vec{x}_b)}{|\vec{x}_a - \vec{x}_b|}\frac{(\vec{x}_a - \vec{x}_b)}{|\vec{x}_a - \vec{x}_b|}$$

  This method is similar to the calculation of spring force. However, for the scaling component, I use the dot product of the position vector and the velocity vector, and then divide by the variation of position.

  Velocity after collision:

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

and

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

  When a collision occurs, it alters the velocity of the sphere and particles. Therefore, we should utilize the formula shown in the figure to update their velocities.Explicit method:

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

Utilize the velocity multiplied by the change in time (delta time) to calculate the next position.

Implicit method:

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

First, estimate the data for the next time step, and then use the velocity at this time to calculate the next position.

Mid point method:

Compute an Euler step

$$\Delta\mathbf{x}=h\cdot f(\mathbf{x}(t_0))$$

Evaluate f at the midpoint

$$f_{mid}=f(\mathbf{x}(t_0)+\frac{\Delta\mathbf{x}}{2})$$

Take a step using the

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

First, estimate the midpoint data, and then use them to calculate the next position.

Runge-Kutta method:

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

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

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

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

First, I follow this procedure to obtain the variables k1, k2, k3, and k4. Then, I use them to calculate the position using the following formulation.

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

- Implementation

  I only post parts of my code in report since it will take lots of place.

  initializeSpring():

  I use several for loops to compute the different types and directions of springs. First, I calculate the length between two particles. Next, according to the annotations, I append the new component into the vector called "_springs". In the new component, it includes the starting index, ending index, length, and type of spring.

```
            }
        }
        for (int i = 0; i < particlesPerEdge - 1; i++) { // vertical
            for (int j = 0; j < particlesPerEdge; j++) {
                int start = i * particlesPerEdge + j;
                int end = (i + 1) * particlesPerEdge + j;
                _springs.emplace_back(start, end, structrualLength, Spring::Type::STRUCTURAL);
            }
        }
        // for shear from left-top to right-bottom
        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 start = i * particlesPerEdge + j;
                int end = (i + 1) * particlesPerEdge + j + 1;
                _springs.emplace_back(start, end, shearLength, Spring::Type::SHEAR);
            }
        }
        // for shear from right-up to left-bottom
        for (int i = 0; i < particlesPerEdge - 1; i++) {
            for (int j = 1; j < particlesPerEdge; j++) {
                int start = i * particlesPerEdge + j;
```

computeSpringForce():

For each component in `_springs`, I follow the formula to calculate the spring force and the damper force. Next, I compute the variation of acceleration and add it to the starting particle's acceleration while subtracting it from the ending particle's acceleration.

```
    for (int i = 0; i < _springs.size(); i++) {
        int start = _springs[i].startParticleIndex();
        int end = _springs[i].endParticleIndex();
        float initialLength = _springs[i].length();
        Eigen::Ref<Eigen::Vector4f> ps = _particles.position(start);  // get start position
        Eigen::Ref<Eigen::Vector4f> pe = _particles.position(end);    // get end position
        Eigen::Vector4f vs = _particles.velocity(start);        //get start velocity
        Eigen::Vector4f ve = _particles.velocity(end);      //get end velocity
        Eigen::Vector4f sub = ps - pe;
        Eigen::Vector4f spring_force = -1 * springCoef * (sub.norm() - initialLength) * sub.normalized();
        Eigen::Vector4f damper_force = -1 * damperCoef * ((vs - ve).dot(sub)) * sub.normalized() / (sub.norm());
        Eigen::Vector4f as = (spring_force + damper_force) * _particles.inverseMass(start);
        Eigen::Vector4f es = (spring_force + damper_force) * _particles.inverseMass(end);
        _particles.acceleration(start) += as;
        _particles.acceleration(end) -= es;
    }
}
```

collide():

For each particle, if the distance between it and the center of the sphere is equal to or smaller than the radius of the sphere, a collision would occur. When a collision happens, I adjust the velocity and position of the particle accordingly. For velocity adjustment, I follow the formula, and for position adjustment, I add a small distance along with the direction from the center of the sphere to the particle to ensure that the sphere won't pass through the cloth.

```cpp
void Spheres::collide(Cloth* cloth) {
  constexpr float coefRestitution = 0.0f;
  // TODO: Collide with particle (Simple approach to handle softbody collision)
  //   1. Detect collision.
  //   2. If collided, update impulse directly to particles' velocity
  // Note:
  //   1. There are `sphereCount` spheres (sphereCount is 1 in the default scene).
  //   2. There are `particlesPerEdge * particlesPerEdge` particles.
  //   3. See TODOs in Cloth::computeSpringForce if you don't know how to access data.
  for (int j = 0; j < particlesPerEdge; j++) {
    for (int x = 0; x < particlesPerEdge; x++) {
      int index = j * particlesPerEdge + x;       // get the specific particle index
      Eigen::Vector4f p_pos = cloth->particles().position(index);       // get the specific particle positio
      if ((p_pos - _particles.position(0)).norm() <= _radius[0]) { // generate collide
        float cm = cloth->particles().mass(index);
        float shm = _particles.mass(0);
        Eigen::Vector4f ua = cloth->particles().velocity(index);
        Eigen::Vector4f ub = _particles.velocity(0);
        cloth->particles().position(index) = _particles.position(0) + (p_pos - _particles.position(0)).norma
        cloth->particles().velocity(index) = (cm * ua + shm * ub + shm * coefRestitution * (ub - ua)) / (cm -
        _particles.velocity(0) = (cm * ua + shm * ub + cm * coefRestitution * (ua - ub)) / (cm + shm);
      }
    }
```

Integrate():

In explicit euler integrator, I use acceleration * delta time to get new velocity, and use velocity * delta time to get new position.

```cpp
void ExplicitEuler::integrate(const std::vector<Particles *> &particles, std::function<void(void)>) const {
  // TODO: Integrate velocity and acceleration
  //   1. Integrate velocity.
  //   2. Integrate acceleration.
  //   3. You should not compute position using acceleration. Since some part only update velocity. (e.g. impulse)
  // Note:
  //   1. You don't need the simulation function in explicit euler.
  //   2. You should do this first because it is very simple. Then you can chech your collision is correct or not.
  //   3. This can be done in 5 lines. (Hint: You can add / multiply all particles at once since it is a large matrix.)
  for (int i = 0; i < particles.size(); i++) {
    particles[i]->position() += deltaTime * particles[i]->velocity();
    particles[i]->velocity() += deltaTime * particles[i]->acceleration();
  }

}
```

In implicit euler integrator, I use same way to get the data, but I use simulateOneStep with new position and velocity to generate other velocity and acceleration, and calculate the result by these.

```cpp
void ImplicitEuler::integrate(const std::vector<Particles *> &particles,
                              std::function<void(void)> simulateOneStep) const {
  // TODO: Integrate velocity and acceleration
  //   1. Backup original particles' data.
  //   2. Integrate velocity and acceleration using explicit euler to get Xn+1.
  //   3. Compute refined Xn+1 using (1.) and (2.).
  // Note:
  //   1. Use simulateOneStep with modified position and velocity to get Xn+1.
  for (int i = 0; i < particles.size(); i++) {
    Eigen::Matrix4Xf v = particles[i]->velocity();//backup
    Eigen::Matrix4Xf a = particles[i]->acceleration(); // backup
    Eigen::Matrix4Xf p = particles[i]->position();  //backup
    // explicit euler
    particles[i]->position() += deltaTime * particles[i]->velocity();
    particles[i]->velocity() += deltaTime * particles[i]->acceleration();
    simulateOneStep(); // get new velocity and acceleration
    particles[i]->position() = p + deltaTime * particles[i]->velocity();
    particles[i]->velocity() = v + deltaTime * particles[i]->acceleration();
  }

}
```

In midpoint euler integrator, it's similar to implicit euler, but the operation before simulatOneStep delta time needs to be divided by 2. Finally use new velocity and acceleration to compute the result.

```cpp
void MidpointEuler::integrate(const std::vector<Particles *> &particles,
                              std::function<void(void)> simulateOneStep) const {
  // TODO: Integrate velocity and acceleration
  //   1. Backup original particles' data.
  //   2. Integrate velocity and acceleration using explicit euler to get Xn+1.
  //   3. Compute refined Xn+1 using (1.) and (2.).
  // Note:
  //   1. Use simulateOneStep with modified position and velocity to get Xn+1.
  for (int i = 0; i < particles.size(); i++) {
    Eigen::Matrix4Xf v = particles[i]->velocity();     // backup
    Eigen::Matrix4Xf a = particles[i]->acceleration(); // backup
    Eigen::Matrix4Xf p = particles[i]->position();     // backup
    particles[i]->position() += deltaTime / 2 * v;
    particles[i]->velocity() += deltaTime / 2 * a;
    simulateOneStep(); // get new velocity and acceleration
    particles[i]->position() = p + deltaTime * particles[i]->velocity();
    particles[i]->velocity() = v + deltaTime * particles[i]->acceleration();
  }
}
```

In Runge Kutta Fourth integrator, I use the formula on ppt to compute the k1, k2, k3 and k4 respectively. I store the position and velocity and use simulateOneStep to calculate velocity and acceleration at that point, and use the results to compute the k1, k2, k3 and k4 of velocity and position.

```
for (int i = 0; i < particles.size(); i++) {
    Eigen::Matrix4Xf v = particles[i]->velocity();      // backup
    Eigen::Matrix4Xf a = particles[i]->acceleration();  // backup
    Eigen::Matrix4Xf p = particles[i]->position();      // backup
    //compute k1
    Eigen::Matrix4Xf k1 = deltaTime * v;
    Eigen::Matrix4Xf k1a = deltaTime * a;
    //compute k2
    particles[i]->position() = p + k1 / 2;
    particles[i]->velocity() = v + k1a / 2;
    simulateOneStep();
    Eigen::Matrix4Xf k2 = deltaTime * particles[i]->velocity();
    Eigen::Matrix4Xf k2a = deltaTime * particles[i]->acceleration();
    //compute k3
    particles[i]->position() = p + k2 / 2;
    particles[i]->velocity() = v + k2a / 2;
    simulateOneStep();
    Eigen::Matrix4Xf k3 = deltaTime * particles[i]->velocity();
    Eigen::Matrix4Xf k3a = deltaTime * particles[i]->acceleration();
    //compute k4
```

- Result and Discussion
  - The difference between integrators

    The major difference between each integrator is the speed at which the cloth drops. This is due to the number of total steps in the integrators. For example, comparing Runge-Kutta Fourth with Explicit Euler, when simulateOneStep() is called, it has to compute force and collide again, which makes it slower to provide the next position's data.
  - Effect of parameters

    springCoef: it is used to compute the spring force of each springs.

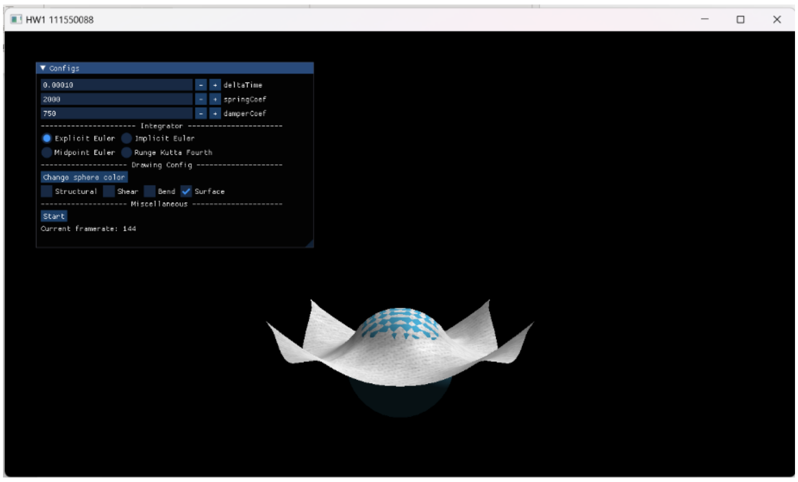
    damperCoef: it is used to compute the damper force of each springs.

    particlesRerEdge:it defines the width and height if we consider the particles on the cloth like a matrix.

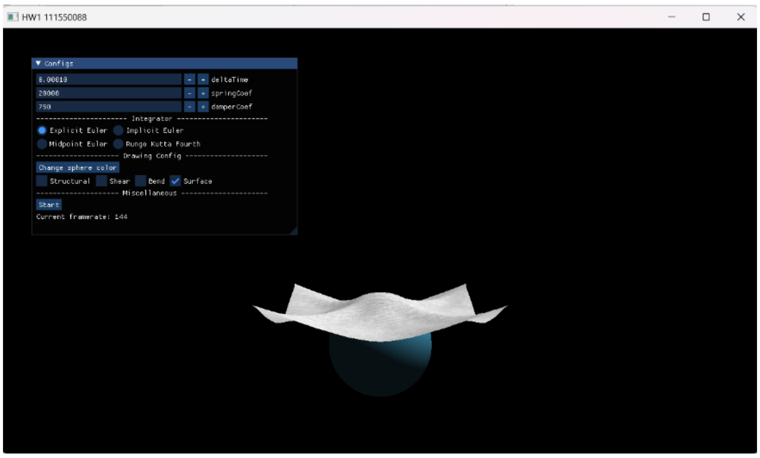
    deltaTime:it’s the time between two calculations of integrate function.

    CoefRestitution:it is used to help compute the velocity after collision.
- Conclusion

  When the spring coefficient is lowered, the cloth's flexibility improves, but its resilience decreases. When the spring coefficient is too small, the cloth behaves like clay, and when the coefficient is too large, it behaves like cardboard.




Set springCoef in 2000 and damperCoef in 750




Set springCoef in 20000 and damperCoef in 750

When the damper coefficient is decreased, the cloth's resilience improves, but the maximum amplitude remains the same. When the damper coefficient is too large, the cloth may disappear, and when it is too low, the cloth may go out of control, resembling an explosion. (I haven't posted my screenshot because it's too fast to capture the scene.)

When the delta time is reduced, the movement of the cloth slows down because the computer has to perform more calculations. Conversely, when the delta time is increased, the cloth may explode due to larger time steps causing instability in the simulation.