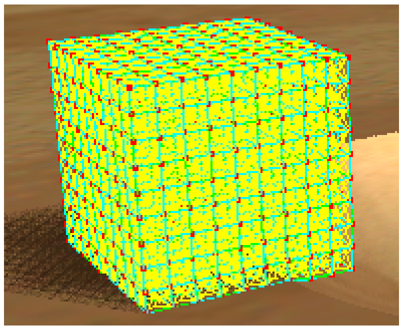
Soft Sim 110550047

Introduction/Motivation

This project practice four integrating methods to simulate particles' behaviors. For the particle object, I construct a complex net struct of soft sim. It contains three connection types of spring, **stretch** (connect to neighbor), **shear** (connect to surrounding) and **bending** (connect to the neighbor of neighbor). In each spring struct, we take two endpoints and treat them as particles, and the inner force is respect to **spring force** and **damper force**. For the integrating method, we implement **Explicit Euler method**, **Implicit Euler method**, **Midpoint Euler method** and **Ronge Kutta Fourth method**. These methods simulate the next motion of each particles, depending on the force field, coefficient, terrain, inner force.

Fundamentals

Spring



- Stretch : Blue Line
- Shear : Yellow Line
- Bending : Green Line
- Each type of connection has their own spring coefficient and damper coefficient.

Force

- Spring Force
  $\vec{f}_a = -k_s(|\vec{x}_a - \vec{x}_b| - r)\frac{\vec{x}_a - \vec{x}_b}{|\vec{x}_a - \vec{x}_b|}$
- Damper Force
  $\vec{f}_a = -k_d(|\vec{v}_a - \vec{v}_b|) \cdot (|\vec{x}_a - \vec{x}_b|)\frac{\vec{x}_a - \vec{x}_b}{|\vec{x}_a - \vec{x}_b|^2}$
- Internal Force
  Update two endpoint and be care of the force direction.

  ```
  particles[spring.getSpringStartID()].addForce(springForce + damperForce);
  particles[spring.getSpringEndID()].addForce(-springForce - damperForce);
  ```

Terrain

- Collision Detection

$$||N|| = 1$$
$$\vec{N} \cdot (\vec{x} - \vec{p}) < \epsilon$$
$$\vec{N} \cdot \vec{v} < 0$$

- Collision Response : Quick modification

$$v' = -k_r v_N + v_T$$

- Contact Detection

$$||N|| = 1$$
$$|N \cdot (x - p)| < \epsilon$$
$$|N \cdot v| < \epsilon$$

- Contact Forces : Add Force

$$f^c = -(N \cdot f)N \quad if\ N \cdot f < 0$$
$$f^f = -k_f(-N \cdot f)v_t \quad if\ N \cdot f < 0$$

Integrator

- **Explicit Euler method**

  x_{n+1} = x_n + ΔV_n

  v_{n+1} = v_n + ΔA_n

```
jelly->getParticle(i).addPosition(jelly->getParticle(i).getVelocity() *
particleSystem.deltaTime);
jelly->getParticle(i).addVelocity(jelly->getParticle(i).getAcceleration() *
particleSystem.deltaTime);
```

- **Implicit Euler method**

  x_{n+1} = x_n + ΔV_{n+1}

  v_{n+1} = v_n + ΔA_{n+1}

```
particleSystem.computeJellyForce(*jelly);
for (int i = 0; i < jelly->getParticleNum(); i++) {
            jelly->getParticle(i).setPosition(origin[i].getPosition() +
particleSystem.deltaTime * jelly->getParticle(i).getVelocity());
            jelly->getParticle(i).setVelocity(origin[i].getVelocity() +
particleSystem.deltaTime * jelly->getParticle(i).getAcceleration());
}
```

- **Midpoint Euler method**

$X_{n+\frac{1}{2}} = x_n + \frac{1}{2}\Delta V_{n+1}$

$V_{n+\frac{1}{2}} = v_n + \frac{1}{2}\Delta A_{n+1}$s

$x_{n+1} = x_n + \Delta V_{n+\frac{1}{2}}$

$v_{n+1} = v_n + \Delta A_{n+\frac{1}{2}}$

```
for (int i = 0; i < jelly->getParticleNum(); i++) {
            origin.push_back(jelly->getParticle(i));
            //Eigen::Vector3f deltaX = 0.5 * particleSystem.deltaTime * jelly-
>getParticle(i).getVelocity();
            Eigen::Vector3f eulerV = jelly->getParticle(i).getAcceleration() *
particleSystem.deltaTime;
            Eigen::Vector3f eulerP = jelly->getParticle(i).getVelocity() *
particleSystem.deltaTime;
            jelly->getParticle(i).addVelocity(eulerV * 0.5);
            jelly->getParticle(i).addPosition(eulerP * 0.5);
            jelly->getParticle(i).setForce(Eigen::Vector3f::Zero());
}
        particleSystem.computeJellyForce(*jelly);
for (int i = 0; i < jelly->getParticleNum(); i++) {
            jelly->getParticle(i).setPosition(origin[i].getPosition() +
particleSystem.deltaTime * jelly->getParticle(i).getVelocity());
            jelly->getParticle(i).setVelocity(origin[i].getVelocity() +
particleSystem.deltaTime * jelly->getParticle(i).getAcceleration());
            jelly->getParticle(i).setForce(Eigen::Vector3f::Zero());
}
```

- **Ronge Kutta Fourth method**

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

$$k_1 = hf(x_0, t_0)$$
$$k_2 = hf(x_0 + \frac{k_1}{2}, t_0 + \frac{h}{2})$$
$$k_3 = hf(x_0 + \frac{k_2}{2}, t_0 + \frac{h}{2})$$
$$k_4 = hf(x_0 + k_3, t_0 + h)$$

```
for (int i = 0; i < jelly->getParticleNum(); i++) {
            StateStep k;
            k.deltaVel = particleSystem.deltaTime * jelly-
>getParticle(i).getAcceleration();
            k.deltaPos = particleSystem.deltaTime * jelly-
>getParticle(i).getVelocity();
```

```
            jelly->getParticle(i).setPosition(origin[i].getPosition() +
particleSystem.deltaTime * jelly->getParticle(i).getVelocity() / 2);
            jelly->getParticle(i).setVelocity(origin[i].getVelocity() +
particleSystem.deltaTime * jelly->getParticle(i).getAcceleration() / 2);
            k1.push_back(k);

        }
        particleSystem.computeJellyForce(*jelly);
        for (int i = 0; i < jelly->getParticleNum(); i++) {
            StateStep k;
            k.deltaVel = particleSystem.deltaTime * jelly-
>getParticle(i).getAcceleration();
            k.deltaPos = particleSystem.deltaTime * jelly-
>getParticle(i).getVelocity();
            jelly->getParticle(i).setPosition(origin[i].getPosition() +
particleSystem.deltaTime * jelly->getParticle(i).getVelocity() / 2);
            jelly->getParticle(i).setVelocity(origin[i].getVelocity() +
particleSystem.deltaTime * jelly->getParticle(i).getAcceleration() / 2);
            k2.push_back(k);
        }
        particleSystem.computeJellyForce(*jelly);
        for (int i = 0; i < jelly->getParticleNum(); i++) {
            StateStep k;
            k.deltaVel = particleSystem.deltaTime * jelly-
>getParticle(i).getAcceleration();
            k.deltaPos = particleSystem.deltaTime * jelly-
>getParticle(i).getVelocity();
            jelly->getParticle(i).setPosition(origin[i].getPosition() +
particleSystem.deltaTime * jelly->getParticle(i).getVelocity());
            jelly->getParticle(i).setVelocity(origin[i].getVelocity() +
particleSystem.deltaTime * jelly->getParticle(i).getAcceleration());
            k3.push_back(k);
        }
        particleSystem.computeJellyForce(*jelly);
        for (int i = 0; i < jelly->getParticleNum(); i++) {
            StateStep k;
            k.deltaVel = particleSystem.deltaTime * jelly-
>getParticle(i).getAcceleration();
            k.deltaPos = particleSystem.deltaTime * jelly-
>getParticle(i).getVelocity();
            k4.push_back(k);
        }
        for (int i = 0; i < jelly->getParticleNum(); i++) {
            jelly->getParticle(i).setPosition(origin[i].getPosition() +
(k1[i].deltaPos + 2 * k2[i].deltaPos + 2 * k3[i].deltaPos + k4[i].deltaPos) /
6);
            jelly->getParticle(i).setVelocity(origin[i].getVelocity() +
(k1[i].deltaVel + 2 * k2[i].deltaVel + 2 * k3[i].deltaVel + k4[i].deltaVel) /
6);
            jelly->getParticle(i).setForce(Eigen::Vector3f::Zero());
        }
```

Result and Discussion

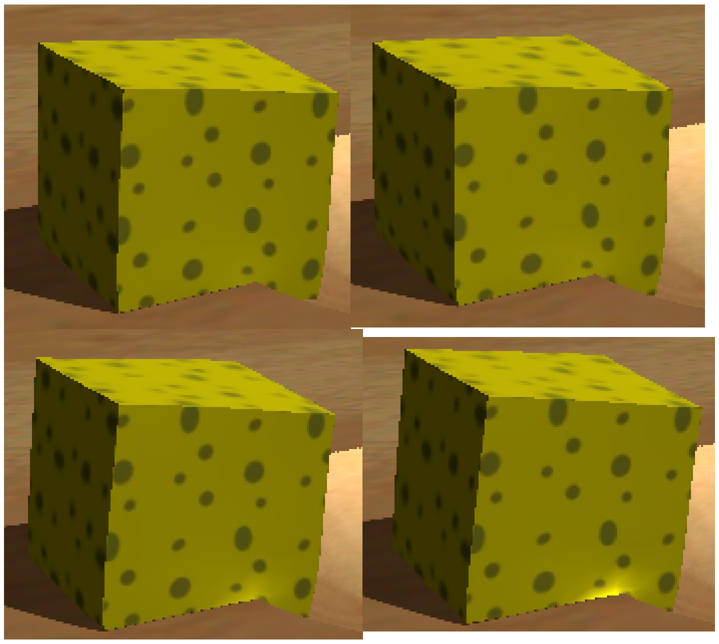
The difference between integrators

Below Screenshot will keep this format.

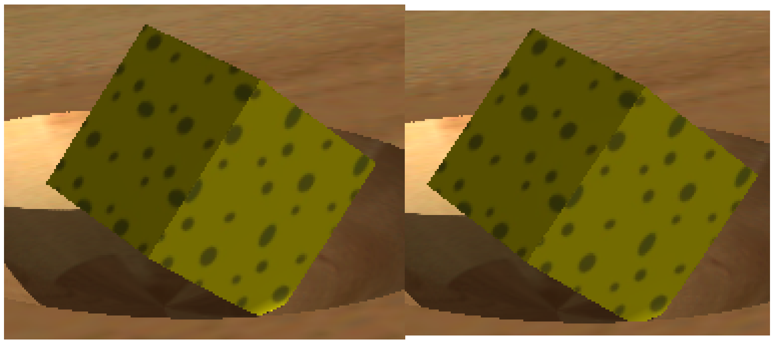
Explicit Euler	**Implicit Euler**
Midpoint	**Ronge Kutta**

- First Bounce

  Midpoint and **Ronge Kutta Fourth** method show the more deformation value at local range, while **Explicit Euler** and **Implicit Euler** method look like alleviating the impact force with the global range.

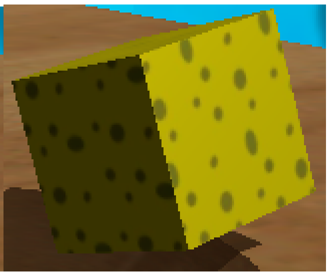
  

- Slide Angle
  Because the previous difference, the angle of sim to slide is distinct. **Explicit Euler** and **Implicit Euler** slide with its edge, while **Midpoint** and **Ronge Kutta Fourth** slide with its face.

- First Touch Margin

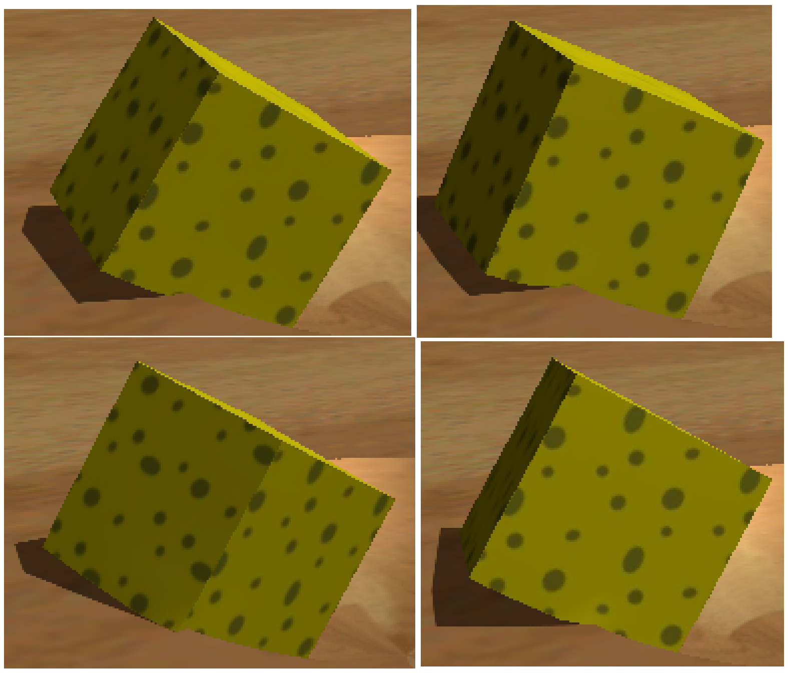
  Explicit Euler and **Implicit Euler** slide out and quickly fall back, the **Midpoint** one stay a bit longer time then falling back, and the **Ronge Kutta Fourth** almost stay in there but eventually fall back to hole.



- Second Touch Margin

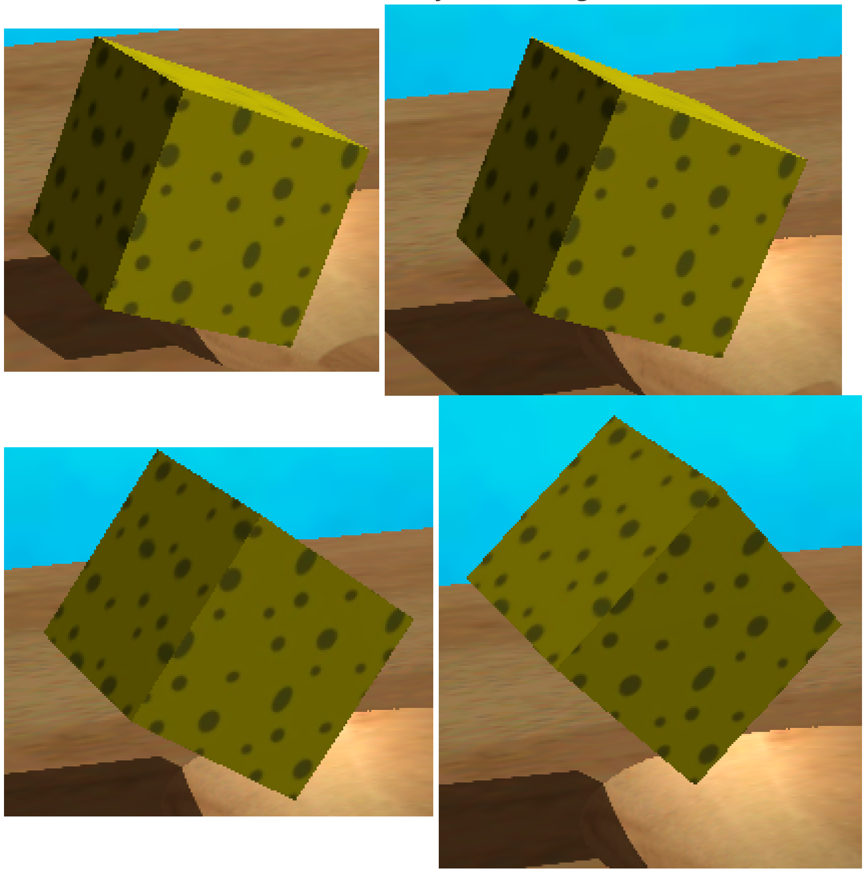
  The previous one apply different force to make sim rotate around y-axis. The **Explicit Euler** one seems stably, while the **Implicit Euler** one turns more angle and suffer from more contact angel so that hardly touch the same height as previous one.

  The **Midpoint** turns approximately 30 degree, and the **Ronge Kutta Fourth** turns almost 90 degree.



Effect of parameters

- Spring Coefficient
  1000、2000、4000、8000, obviously bounce higher



- Damper Coefficient
  10

60

- Resist Coefficient

0.8 : 0.2 :

- Friction Coefficient

0.2 : 0.8 :

- Jelly Rotation

Deformation Duration

Conclusion

In this project, we simulate many force and influence factors to show the difference between integrate methods. In general, adjust the spring coefficient and damper coefficient much lower and enhance friction coefficient will simulate just like a real sim there. Depending on situations, we can choose different method to implement.
Explicit is much simple but not seems real at local part.
Implicit also not seems real at local part, but is much reasonable in this project.
Midpoint can observe the deformation and is the most suitable method in this method.
Ronge Kutta can observe the deformation at local part, but it may be much stubborn and heavy when simulating the sim.
Above all, not every situation has the commonly best answer. If so, we also need to simulate every part as a real world, such as the small force inside, but it is not essential and necessary. Under the simulation, we can trial and error and see the much closer to the reality or imagine one.