CASE HW1

309553008

1. Introduction:

   這是模擬鋼體物件碰撞的情境，骰子從高空落下，碰撞到不同物體應該跟形狀、方向、速度……不同，而有不同反應。

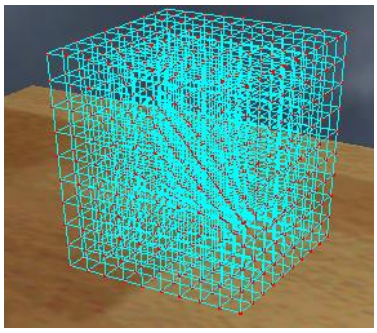
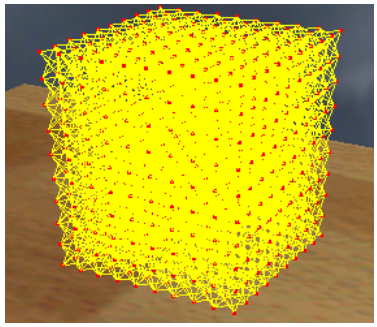
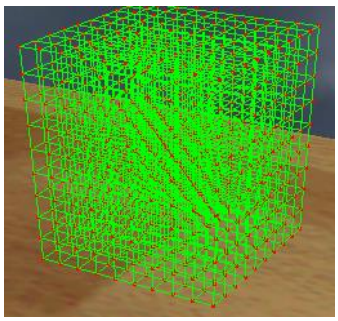
2. Fundamentals:

   骰子有三種不同的彈簧，依照外力(我看起來都是重力與碰觸的反作用力還可能有磨擦力)不同可以求出速度，有了速度後可以估算出物體下個瞬間(delta_T)的位置。

3. Implementation:

   Cube:

   在 initializeSpring()裡去實作 3 種不同的彈簧(把 particle 連起來)
   分別為 stretch / shear / bend

   在 computeSpringForce()與 computeDamperForce()裡分別去計算彈簧力與恢復力，最後加在 computeInternalForce()裡去計算每一個 particle 是否為某彈簧的端點，如果是的話就把該彈簧力加到該 particle 上

```
void Cube::computeInternalForce() {
  // TODO
  for (int i = 0; i < particles.size(); i += 1)
  {
    for (int j = 0; j < springs.size(); j += 1)
    {
      if (springs[j].getSpringStartID() == i)
      {
        particles[i].addForce(computeSpringForce(particles[i].getPosition(), particles[spri
        particles[i].addForce(computeDamperForce(particles[i].getPosition(), particles[sp
      }
      else if (springs[j].getSpringEndID() == i)
      {
        particles[i].addForce(computeSpringForce(particles[i].getPosition(), particles[spri
        particles[i].addForce(computeDamperForce(particles[i].getPosition(), particles[sp
      }
    }
  }
```

Terrain:

根據不同的場景做不同的碰撞操作

Plane / TiltedPlane:

先對 particle 到平面的向量與 normal 做內積，計算距離，若小於給定的 epsilon 那就是很接近平面(或貼著)，若在這狀況下速度 V 與 normal 的內積小於 0 的話就是會碰撞(角度小於 90 度，所以前進方向是向平面的)，然後照著 ppt 的公式分別算出碰撞後的速度、摩擦力與反作用力。

```
Eigen::Vector3f N = this->normal / this->normal.norm();
for (int i = 0; i < cube.getParticleNum(); i += 1)
{
    if (abs(N.dot(cube.getParticle(i).getPosition() - this->position)) < eEPSILON) // on the wall
    {
        if (N.dot(cube.getParticle(i).getVelocity()) < 0)  // collision
        {
            Eigen::Vector3f Vn = cube.getParticle(i).getVelocity().dot(N) * N;
            Eigen::Vector3f Vt = cube.getParticle(i).getVelocity() - Vn;
            Eigen::Vector3f V_after = -1 * coefResist * Vn + Vt;
            cube.getParticle(i).setVelocity(V_after);

            Eigen::Vector3f Fc = -(N.dot(cube.getParticle(i).getForce()) * N);
            cube.getParticle(i).addForce(Fc);

            Eigen::Vector3f Ff = -coefFriction * (-N.dot(cube.getParticle(i).getForce()) * Vt);
            cube.getParticle(i).addForce(Ff);
        }
    }
}
```

Sphere:

計算 particle 到球心的距離減掉半徑是否小於 epsilon，若是小於的話就是會碰撞，之後依照 ppt 的公式計算出碰撞後的速度、反作用力與摩擦力

```
for (int i = 0; i < cube.getParticleNum(); i += 1)
{
    Particle p = cube.getParticle(i);
    if ((p.getPosition() - this->position).norm() - this->radius < eEPSILON) // collision
    {
        Eigen::Vector3f N = (cube.getParticle(i).getPosition() - this->position) / (cube.getParticle(i).getPosition() - this->position).norm();
        if (N.dot(cube.getParticle(i).getVelocity()) < 0)  // collision
        {
            Eigen::Vector3f Vn = cube.getParticle(i).getVelocity().dot(N) * N;
            Eigen::Vector3f Vt = cube.getParticle(i).getVelocity() - Vn;
            Eigen::Vector3f V_after = Vn * (cube.getParticle(i).getMass() - this->mass) / (cube.getParticle(i).getMass() + this->mass) + Vt;
            cube.getParticle(i).setVelocity(V_after);

            Eigen::Vector3f Fc = -(N.dot(cube.getParticle(i).getForce()) * N);
            cube.getParticle(i).addForce(Fc);

            Eigen::Vector3f Ff = -coefFriction * (-N.dot(cube.getParticle(i).getForce()) * Vt);
            cube.getParticle(i).addForce(Ff);
        }
    }
}
```

Bowl 與 Sphere 差不多，只是 normal 方向不一樣，與判斷碰撞的條件不一樣(變成半徑減掉 particle 到球心的距離是否小於 epsilon)

Integrator 基本上都是實作老師上課講義裡的公式

- ExplicitEuler:

  利用現有的速度更新位置

```
void ExplicitEulerIntegrator::integrate(MassSpringSystem& particleSystem) {
  // TODO
  for (int i = 0; i < particleSystem.getCubeCount(); i += 1) {
    for (int j = 0; j < particleSystem.cubes[i].getParticleNum(); j += 1) {
      Eigen::Vector3f force = particleSystem.cubes[i].getParticle(j).getForce();
      Eigen::Vector3f velocity = particleSystem.cubes[i].getParticle(j).getVelocity();
      Eigen::Vector3f position = particleSystem.cubes[i].getParticle(j).getPosition();
      float mass = particleSystem.cubes[i].getParticle(j).getMass();
      float deltatime = particleSystem.deltaTime;

      // newpos = origin_pos + vt
      position += velocity * deltatime;
      // f = ma, v = at, v = ft/m
      velocity += force * deltatime / mass;
      // set new pos and v
      particleSystem.cubes[i].getParticle(j).setVelocity(velocity);
      particleSystem.cubes[i].getParticle(j).setPosition(position);
```

- ImplicitEuler:

  先更新完速度後(也就是下個時間點的 V)，利用該速度再去更新位置

```
void ImplicitEulerIntegrator::integrate(MassSpringSystem& particleSystem) {
  // TODO
  for (int i = 0; i < particleSystem.getCubeCount(); i += 1)
  {
    particleSystem.computeCubeForce(particleSystem.cubes[i]);
    for (int j = 0; j < particleSystem.cubes[i].getParticleNum(); j += 1)
    {
      Eigen::Vector3f force = particleSystem.cubes[i].getParticle(j).getForce();
      Eigen::Vector3f velocity = particleSystem.cubes[i].getParticle(j).getVelocity();
      Eigen::Vector3f position = particleSystem.cubes[i].getParticle(j).getPosition();
      float mass = particleSystem.cubes[i].getParticle(j).getMass();
      float delta_time = particleSystem.deltaTime;

      //  f = ma, v = v0 + at, v = v0 + f / m * t
      velocity = force / mass * delta_time;
      //  newpos = oripos + vt
      position += velocity * delta_time;

      particleSystem.cubes[i].getParticle(j).setVelocity(velocity);
      particleSystem.cubes[i].getParticle(j).setPosition(position);
    }
  }
```

- MidpointEuler:
  利用現在與下個時間點中間的那個時間點(也就是 t0 + delta_T/2)的速度去更新位置

```
for (int i = 0; i < particleSystem.getCubeCount(); i += 1)
{
    for (int j = 0; j < particleSystem.cubes[i].getParticleNum(); j += 1)
    {
        Eigen::Vector3f force = particleSystem.cubes[i].getParticle(j).getForce();
        Eigen::Vector3f velocity = particleSystem.cubes[i].getParticle(j).getVelocity();
        Eigen::Vector3f position = particleSystem.cubes[i].getParticle(j).getPosition();
        float mass = particleSystem.cubes[i].getParticle(j).getMass();
        float delta_time = particleSystem.deltaTime;

        //velocity += delta_time / 2 * particleSystem.cubes[i].getParticle(j).getAcceleration();
        velocity += delta_time / 2 * force / mass;
        position += delta_time * velocity;
        //  f = ma, v = v + at, v = v + f / m * t
        velocity = particleSystem.cubes[i].getParticle(j).getVelocity() + delta_time * force / mass ;

        particleSystem.cubes[i].getParticle(j).setVelocity(velocity);
        particleSystem.cubes[i].getParticle(j).setPosition(position);
    }
}
```

- RungeKuttaFourth:
  利用不同時間點的速度去計算不同的預測位置(k1~4)然後再對這些位置做加權去更新位置

```
for (int i = 0; i < particleSystem.getCubeCount(); i += 1)
{
    for (int j = 0; j < particleSystem.cubes[i].getParticleNum(); j += 1)
    {
        Eigen::Vector3f force = particleSystem.cubes[i].getParticle(j).getForce();
        Eigen::Vector3f velocity = particleSystem.cubes[i].getParticle(j).getVelocity();
        Eigen::Vector3f position = particleSystem.cubes[i].getParticle(j).getPosition();
        float mass = particleSystem.cubes[i].getParticle(j).getMass();
        float delta_time = particleSystem.deltaTime;

        StateStep s;
        s.deltaVel = Eigen::Vector3f(0, 0, 0);
        s.deltaPos = Eigen::Vector3f(0, 0, 0);
        Eigen::Vector3f V = Eigen::Vector3f(0, 0, 0);
        Eigen::Vector3f P = Eigen::Vector3f(0, 0, 0);
        // k1
        s.deltaVel += velocity + force / mass * delta_time;
        s.deltaPos = position + s.deltaVel * delta_time;
        V += s.deltaVel;
        P += s.deltaPos;
        // k2
        s.deltaVel += s.deltaVel * delta_time/2;
        s.deltaPos = position + delta_time * s.deltaVel;
        V += s.deltaVel * 2;
        P += s.deltaPos * 2;
        // k3
        s.deltaVel += s.deltaVel * delta_time / 2;
        s.deltaPos = position + delta_time * s.deltaVel;
        V += s.deltaVel * 2;
        P += s.deltaPos * 2;
        // k4
        s.deltaVel += s.deltaVel * delta_time;
        s.deltaPos = position + delta_time * s.deltaVel;
        V += s.deltaVel;
        P += s.deltaPos;

        particleSystem.cubes[i].getParticle(j).setVelocity(V/6);
        particleSystem.cubes[i].getParticle(j).setPosition(P/6);
    }
}
```

4. Result and Discussion / Conclusion:

- The difference between integrators:
  因為電腦的算力比較不好，所以都沒甚麼比較明顯差異的感覺，差別比較大的是用 RK4 的話在第一次反射會明顯的反彈的比較高，然後用 ImplicitEuler 則是算得比較久
- Effect of parameters:
  Delta Time 如果太小的話 position 會更新很慢，然後因為算力不足的話看起來根本沒在動，如果太大則會導致程式有地方 carsh(雖然我目前想不太到為什麼，我以為應該位子更新幅度比較大，所以可以穿模會比較明顯)
- Spring/ Damper Coef:
  越大的話彈簧越硬，反之則反，感覺上 Damper Coef 是彈簧內部的彈力，而 Spring Coef 則比較跟其他物體發生碰撞的彈力有關係

這份 code 的架構真的有點大(複雜)，所以看很久，像是有些接口根本不知道要怎麼接，哪邊的參數是哪邊傳來的還是本身這個 class 的 private 參數?然後有些 TODO 的地方我實在是不知道要幹嘛，像是以 computeInternalForce()這個 function 為例，雖然看名字知道要算內力，但是要算甚麼內力、怎麼算、要 call 什麼 function 算嗎，還是直接手刻呢?算完後要怎麼存?又要 call 哪個 function 來存到哪個結構?當然也可能是我太爛了吧 qq

然後比較想提的是，我有遇到一個不明所以的 bug，我看 Eigen 這個 liberary 中的介紹是說 norm()是對每一維平方後加起來算平方根，然後 size()則是直接寫向量長度，所以理論上在這次作業中應該沒有差吧(?)然後我一開始在建彈簧都是用 size()去計算長度，所以後面有很長一段時間我的 collison 一直爆掉，會向下面這樣，但是改成 norm()就什麼問題都沒有了，神秘(?