111550149 林悅揚

Introduction/Motivation

本報告旨在探討不同數值積分方法對粒子系統模擬（特別是布料和剛體球體）的影響。我們用 C++和 Visual Studio2020 實作了顯式歐拉法、隱式歐拉法、中點法以及 Runge-Kutta 4 階方法來解析粒子的運動方程，分析了這些方法在模擬動力學行為時的效率和準確性。

Fundamentals

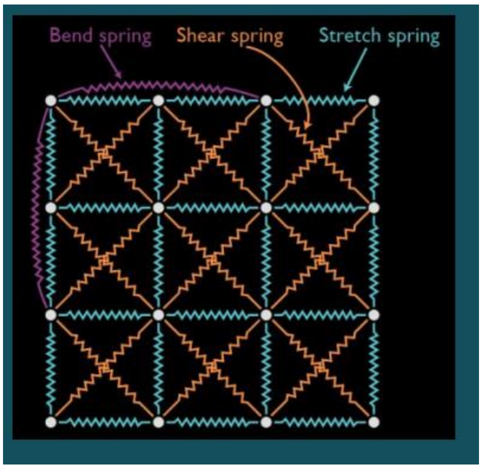
- Compute Spring Force<“particles.pptx” from p.9 - p.13>
- Collision<“particles.pptx” from p.14 - p.19>
- Integrator
  - Explicit Euler<“ODE_basics.pptx” from p.15 - p.16>
  - Implicit Euler<"ODE_implicit.pptx" from p.18 - p.19>
  - Midpoint Method<“ODE_basics.pptx” from p.18 - p.20 and “pbm.pdf” from B.5 - B.6 >
  - RK4<“ODE_basics.pptx” p.21 and “pbm.pdf” from B.5 - B.6 >

Implementation

- **cloth.cpp**

```cpp
void Cloth::initializeSpring() {
  // TODO: Connect particles with springs.
  //   1. Compute spring length per type.
  //   2. Iterate the particles. Push spring objects into `_springs` vector
  // Note:
  //   1. The particles index:
  //   ===============================================
  //   0 1 2 3 ... particlesPerEdge - 1
  //   particlesPerEdge ... ...
  //   ... ... particlesPerEdge * particlesPerEdge - 1
  //   ===============================================
  // Here is a simple example which connects the horizontal structrual springs.
  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);
    }
  }
  float shearlen = (_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, shearlen, Spring::Type::SHEAR);
    }
  }
  for (int i = 0; i < particlesPerEdge - 1; ++i) {
    for (int j = 1; j < particlesPerEdge; ++j) {
      int index = i * particlesPerEdge + j;
      _springs.emplace_back(index, index + particlesPerEdge-1, shearlen, Spring::Type::SHEAR);
    }
  }
```

```cpp
}
float bendlen = (_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, bendlen, 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, bendlen, Spring::Type::BEND);
  }
}
```



跟據給的範例去實作，依照給的範例以及上圖去實作

STRUCTUAL 垂直每行的 index 差 particlesPeredge

SHERA 是跟下一行的+1-1 去連接

BEND 根為右邊第二個和下面第二個連接原理跟 STRUCTUAL 一樣

```
void Cloth::computeSpringForce() {
  // TODO: Compute spring force and damper force for each spring.
  //   1. Read the start and end index from spring
  //   2. Use _particles.position(i) to get particle i's position.
  //   3. Modify particles' acceleration a = F / m;
  // Note:
  //   1. Use _particles.inverseMass(i) to get 1 / m can deal with m == 0. Which will returns 0.
  // Hint:
  //   1. Use a.norm() to get length of a.
  //   2. Use a.normalize() to normalize a inplace.
  //          a.normalized() will create a new vector.
  //   3. Use a.dot(b) to get dot product of a and b.
    for (const auto& spring : _springs) {
        int start = spring.startParticleIndex();
        int end = spring.endParticleIndex();
        //force direct is converse of the position
        //springforce
        Eigen::Vector4f direction = _particles.position(start) - _particles.position(end);  // xa-xb
        float currlen = direction.norm();//|xa-xb|
        direction.normalize();//1 dirc
        float delta_l = currlen - spring.length();
        Eigen::Vector4f springforce = direction * (springCoef * delta_l);
        //damperforce
        Eigen::Vector4f relatev = _particles.velocity(start) - _particles.velocity(end);  //va-vb
        float delta_v = relatev.dot(direction);
        Eigen::Vector4f dampforce =direction * (damperCoef * delta_v);
        _particles.acceleration(start) -= (dampforce + springforce) * _particles.inverseMass(start);
        _particles.acceleration(end) += (dampforce + springforce) * _particles.inverseMass(end);
    }
}
```

根據 ppt 的公式去實作，讓力的方想指向起始點，先計算 spring force

$\vec{F_{spring}} = -k(\Delta l)\hat{d}$ k 為彈簧常數

再去計算 Damper force $\vec{F_{damp}} = -c(\Delta y)\hat{d}$ c 圍阻尼係數

再把最後的力總和所造成的加速度加成在起始點以及結束點

- **Sphere.cpp**

```cpp
void Spheres::collide(Shape* shape) { shape->collide(this); }
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 i = 0; i < sphereCount; i++) {
        for (int j = 0; j < particlesPerEdge * particlesPerEdge; j++) {
        // ditectcollide
        Eigen::Vector4f nor = _particles.position(i) - cloth->particles().position(j);
        float shpclodis = nor.norm();
        //increase radius led the cloth not pentrate the sphere
        if (_radius[i]+0.01 < shpclodis) {
            continue;//no collide
            }
        nor.normalize();
        Eigen::Vector4f relavel = _particles.velocity(i) - cloth->particles().velocity(j);
        Eigen::Vector4f normalvel = nor * relavel.dot(nor);
        //if two get close
        if (relavel.dot(nor) < 0) {
            float invermsph = _particles.inverseMass(i);
            float inversmclo = cloth->particles().inverseMass(j);
            Eigen::Vector4f impulse = -(1+coefRestitution) * normalvel/(invermsph+inversmclo);
            _particles.velocity(i) += impulse * invermsph;
            cloth->particles().velocity(j) -= impulse * inversmclo;
        }
        }
    }
}
```

先處理 detect collision 我的想法是球和計算圓心到布料粒子的距離，

如果這個距離>半徑(+0.01 因為如果不加這個偏移量布料會穿過去，只有布料粒子會停在 sphere 的表面)就代表沒有發生碰撞，反之碰撞了

$J = -(1+e)(\vec{v_{rel}} \cdot \hat{d})/(\frac{1}{m_{sphere}} + \frac{1}{m_{cloth}})$ 依照這個公式去計算衝量，並將其造成的速度變化加成在 sphere 和 cloth

- **integrator.cpp**

```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 (auto &p : particles) {
    //deltatime in config.h
    //change position first or it will get wrong position
    p->position() += deltaTime * p->velocity();
    p->velocity() += deltaTime * p->acceleration();
  }
}
```

Explicit Euler:根據當前的速度和加速度來更新粒子的位址和速度

```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.
    //step1
    std::vector<Particles> backup;
    for (int i = 0; i < particles.size(); ++i) {
      backup.push_back(*particles[i]);  // backup the particle
    }
    // step2
    for (auto &p : particles) {
        p->position() += deltaTime * p->velocity();
        p->velocity() += deltaTime * p->acceleration();
    }
    simulateOneStep();
    // step3
    for (int i = 0; i < particles.size(); ++i) {
        particles[i]->position() = backup[i].position() + particles[i]->velocity() * deltaTime;
        particles[i]->velocity() = backup[i].velocity() + particles[i]->acceleration() * deltaTime;
    }
}
```

Implicit Euler:跟 Explicit Euler 比，這方式考慮了下一個時間的速度和加速度更新粒子的狀態

```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.
  // step1
  std::vector<Particles> backup;
  for (int i = 0; i < particles.size(); ++i) {
    backup.push_back(*particles[i]);  // backup the particle
  }
  simulateOneStep();
  // step2
  for (auto &p : particles) {
    p->position() += 0.5f*deltaTime * p->velocity();
    p->velocity() += 0.5f*deltaTime * p->acceleration();
  }
  // step3
  for (int i = 0; i < particles.size(); ++i) {
    particles[i]->position() = backup[i].position() + particles[i]->velocity() * deltaTime;
    particles[i]->velocity() = backup[i].velocity() + particles[i]->acceleration() * deltaTime;
  }
}
```

Midpoint Method:為一種二階積分方法，這種方式比 Explicit Euler 能提高更好的精度因為他更新粒子的狀態考慮了時間的中間點

```cpp
void RungeKuttaFourth::integrate(const std::vector<Particles *> &particles,
                                 std::function<void(void)> simulateOneStep) const {
    // TODO: Integrate velocity and acceleration
    //   1. Backup original particles' data.
    //   2. Compute k1, k2, k3, k4
    //   3. Compute refined Xn+1 using (1.) and (2.).
    // Note:
    //   1. Use simulateOneStep with modified position and velocity to get Xn+1.
    //backup
    std::vector<Particles> backup;


    for (int i = 0; i < particles.size(); ++i) {
        backup.push_back(*particles[i]);  // backup the particle
    }
    std::vector<Particles> k1(backup);
    std::vector<Particles> k2(backup);
    std::vector<Particles> k3(backup);
    std::vector<Particles> k4(backup);
    //k1
    for (int i = 0; i < backup.size(); ++i) {
        //update the particle
        particles[i]->position() = backup[i].position() + (particles[i]->velocity() * deltaTime * 0.5f);
        particles[i]->velocity() = backup[i].velocity() + (particles[i]->acceleration() * deltaTime * 0.5f);
        //store k1
        k1[i].position() = particles[i]->velocity()*deltaTime;
        k1[i].velocity() = particles[i]->acceleration() * deltaTime;
    }
    simulateOneStep();
    for (int i = 0; i < backup.size(); ++i) {
      // update the particle
      particles[i]->position() = backup[i].position() + (particles[i]->velocity() * deltaTime * 0.5f);
      particles[i]->velocity() = backup[i].velocity() + (particles[i]->acceleration() * deltaTime * 0.5f);
      // store k2
      k2[i].position() = particles[i]->velocity() * deltaTime;
      k2[i].velocity() = particles[i]->acceleration() * deltaTime;
    }
```

```
    }
    simulateOneStep();
    for (int i = 0; i < backup.size(); ++i) {
      // update the particle
      particles[i]->position() = backup[i].position() + (particles[i]->velocity() * deltaTime * 0.5f);
      particles[i]->velocity() = backup[i].velocity() + (particles[i]->acceleration() * deltaTime * 0.5f);
      // store k3
      k3[i].position() = particles[i]->velocity() * deltaTime;
      k3[i].velocity() = particles[i]->acceleration() * deltaTime;
    }
    simulateOneStep();
    for (int i = 0; i < backup.size(); ++i) {
      // store k4
      k4[i].position() = particles[i]->velocity() * deltaTime;
      k4[i].velocity() = particles[i]->acceleration() * deltaTime;
    }
    for (int i = 0; i < backup.size(); ++i) {
      //  Runge-Kutta
      particles[i]->position() =
          backup[i].position() +
          (k1[i].position() + 2.0f * k2[i].position() + 2.0f * k3[i].position() + k4[i].position()) / 6.0f;

      particles[i]->velocity() =
          backup[i].velocity() +
          (k1[i].velocity() + 2.0f * k2[i].velocity() + 2.0f * k3[i].velocity() + k4[i].velocity()) / 6.0f;
    }


}
```

Runge-Kutta 4:通過結合四個不同的斜率(k1k2k3k4)來預測粒子下一個狀態，每個 k 值是基於上一步的信息，將斜率組合計算，得到下一步的位置和速度

$$y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4),$$

$$\begin{aligned}
k_1 &= f(t_i, y_i), \\
k_2 &= f\left(t_i + \frac{h}{2}, y_i + \frac{h}{2}k_1\right), \\
k_3 &= f\left(t_i + \frac{h}{2}, y_i + \frac{h}{2}k_2\right), \\
k_4 &= f(t_i + h, y_i + hk_3),
\end{aligned}$$

Result and Discussion

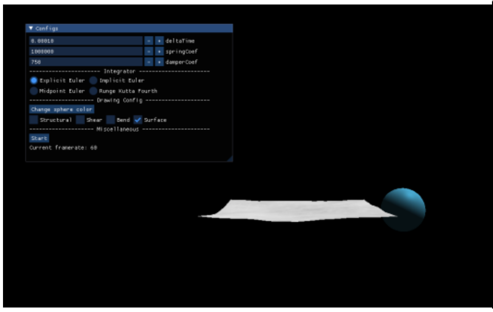
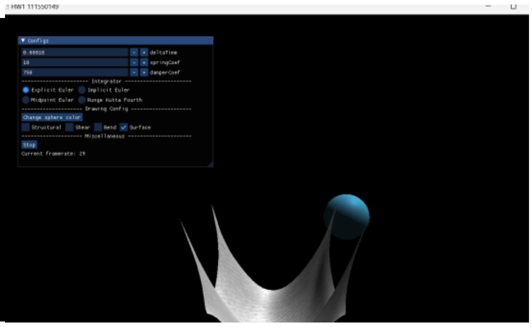
The difference between integrators

我藉由調整 deltatime 來看不同的 integrators 的穩定度，把 deltatime 變大後可以發現 Explicit Euler 會比較不穩定(布有可能會直接爆掉)，此

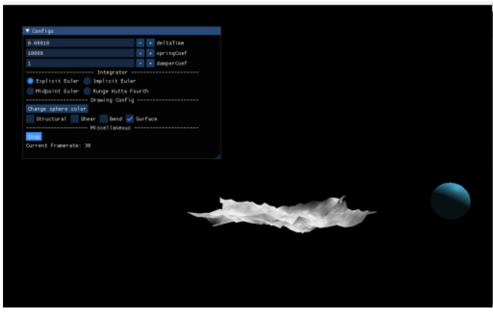
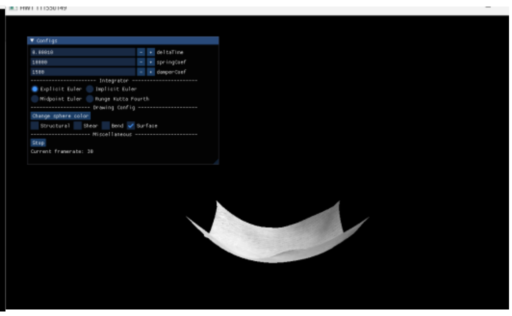
時 Midpoint Method 和 Runge-Kutta 4 顯得穩定很多，但 Implicit Euler 好像有點問題反而是最不穩定的(布直接爆掉)，而 Runge-Kutta 4 會變得很當，畫面會卡卡的

Effect of parameters

首先我們改變 springCoef 可以發現 springCoef 越大，布的可改變性越小太高的 springCoef 布料的響應會過於迅速，從而失去現實感，甚至布會爆掉。反之，過低的彈簧係數則會導致布料的響應過於緩慢，甚至在重力作用下過度垂張，可看下圖(皆是把球移走的情況)。

再來我們改變 damperCoef，他對整對於控制系統中震盪的衰減速度至關重要，過小的 damperCoef 會讓布過度震動而太大的 damperCoef 會讓布很僵硬，一點波動產生的皺褶都沒有，如下圖，皆由移動球讓布產生震動再把球移走。

Result and Discussion

integrators 之間的對比顯示，隨著時間步長的增加，Explicit Euler 的穩定性降低，特別是在處理更具挑戰性的動力學情況時。Midpoint Euler 和 Runge-Kutta 4 階法在大部分情況下提供了更高的穩定性和精確度。然而，Implicit Euler 在某些情況下表現出不預期的不穩定行為，這可能指向了實現中的潛在問題，或者需要對 integrator 參數進行進一步的調優。

在參數效應方面，springCoef 和 damperCoef 對模擬結果有著顯著的影響。springCoef 的增加會提高布料的剛度，進而影響其動態反應，而過高或過低的值均會導致不切實際的物理行為。damperCoef 在控制系統的震盪和提供穩定的動態反應方面發揮著關鍵作用。適當的 damperCoef 可以使布料在撤去外部作用力後迅速穩定下來，而過大或過小的阻尼係數則會導致不自然的運動行為。

綜上所述，我們的研究強調了適當選擇和調整數值 integrator 對於高質量物理模擬的重要性。未來的工作將專注於進一步改進 Implicit Euler 的穩定性，探索自適應時間步長策略，並擴展我們的方法來模擬更加複雜的物理系統。此外，對於模擬參數的細微調整和優化，將在提升模擬真實性方面起到關鍵作用。