Introduction to Computer Animation HW1 Report

109350008 張詠哲

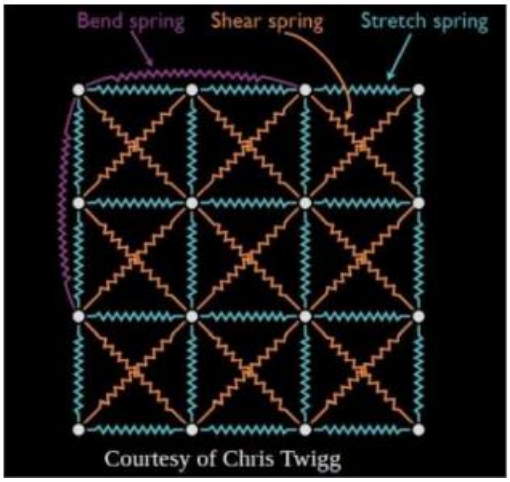
○ Introduction/Motivation

在這次的作業中我們要實作一個粒子系統, 模擬一個蒟蒻(粒子上繫著一堆彈簧)自由落體到一個有碗狀洞的一個平面上得情形。首先連接各個粒子的彈簧種類有 STRUCT、BENDING、SHEAR, 而其中也計算蒟蒻的受力,像是 damper force、Spring force, 以及碰撞到平面以及碗的碰撞模擬, 並且實作了 4 種不同的 integrator, 包含 Explicit Euler Method、 Implicit Euler Method、Midpoint Euler Method、Runge Kutta 4th Method。

○ Fundamentals

- Spring

蒟蒻由一個 10 x 10 x 10 的粒子構成, 其中連接他們的彈簧種類有 struct、bending、shear。 Struct 為連接前後左右上下的鄰近粒子其示而 bending 和 struct 很像,只是他要間隔一個粒子, shear 為連接斜角方向的粒子。示意圖如下:



- Internal force

包含了 Springforce 以及 Damperforce。Spring force 作用是在當發生震動時可以儲存能量、而 Damper force 是為了可以減少震動能量。

- Collision

Collision 分成 2 個部分, 分別是平面以及碗的部分, 要先偵測到碰撞,後要計算當碰撞發生後的速度更新以及受力情形,而受力可以再細分成兩個分別為 contact force 以及 friction force。平面和碗皆為一個不會移動的地形, 其中碗狀洞可以想成一個只有畫下半部且質量為 10 的巨大粒子。

- Integration

▪ Explicit Euler Method

Explicit Euler Method 是一種簡單計算為微分方程的一種辦法, 其主要算法其實也跟我們在國高中物理課計算位置的公式:

$$X_{n+1} = \mathrm{X_n} + \Delta \mathrm{t} * \mathrm{f(X, t)}$$

▪ Implicit Euler Method

Implicit Euler method 和 Explicit Euler method 很像,只是 Explicit Euler 使用的是當下的為微分值, 而 Implicit Euler 是用下一個時間差的位置以及速度來更新現在的位置以及速度。

$$\mathrm{X_{n+1}} = X_n + \Delta t * f(X_{n+1}, \mathrm{t}_{n+1})$$

▪ Midpoint Euler Method

和 Implicit Euler method 很像, 但是 Implicit Euler 是用 1 倍的時間差, 而 Midpoint Euler 是使用 0.5 倍的時間差。

$$\mathrm{X_{n+1}} = \mathrm{X_n} + \Delta \mathrm{t} * \mathrm{f_{mid}}$$

▪ Runge Kutta 4th Method

Runge Kutta 4th method 和 Midpoint Euler method 很像用不同的微分值在使用 Explicit Euler 去計算下一個微分值, 接著把上述算完的微分值各自成上權重, 再計算一次 explicit method 得到最後的解。

○ Implementation

- Jelly

這部分主要只要找出各粒子的 index 以及排列的方式, 再依照各彈簧的要求連上就可以了。

- STRUCT/BENDING:

共有上下左右前後 6 個方向, 但是因為是兩粒子皆各有 6 種方向, 因此, 只要實作 6/2 = 3 種不重複的方向就好。以下是部分的 STRUCT 、BENDING 程式碼, 其餘以此類推

```
// struct
// x-direction
for (int i = 0; i < particleNumPerEdge - 1; i++) {
    for (int j = 0; j < particleNumPerEdge; j++) {
        for (int k = 0; k < particleNumPerEdge; k++) {
            int iParticleID = i * particleNumPerFace + j * particleNumPerEdge + k;
            int iNeighborID = iParticleID + particleNumPerFace;
            Eigen::Vector3f SpringStartPos = particles[iParticleID].getPosition();
            Eigen::Vector3f SpringEndPos = particles[iNeighborID].getPosition();
            Eigen::Vector3f Length = SpringStartPos - SpringEndPos;
            float absLength = sqrt(Length[0] * Length[0] + Length[1] * Length[1] + Length[2] * Length[2]);

            springs.push_back(Spring(iParticleID, iNeighborID, absLength, springCoefStruct, damperCoefStruct,
                                     Spring::SpringType::STRUCT));
        }
    }
}
```

```
// i -> 前, j -> 上, z -> 右
// bending
// z-direction
for (int i = 0; i < particleNumPerEdge; i++) {
    for (int j = 0; j < particleNumPerEdge; j++) {
        for (int k = 0; k < particleNumPerEdge - 3; k++) {
            int iParticleID = i * particleNumPerFace + j * particleNumPerEdge + k;
            int iNeighborID = iParticleID + 3;
            Eigen::Vector3f SpringStartPos = particles[iParticleID].getPosition();
            Eigen::Vector3f SpringEndPos = particles[iNeighborID].getPosition();
            Eigen::Vector3f Length = SpringStartPos - SpringEndPos;
            float absLength = sqrt(Length[0] * Length[0] + Length[1] * Length[1] + Length[2] * Length[2]);

            springs.push_back(Spring(iParticleID, iNeighborID, absLength, springCoefStruct, damperCoefStruct,
                                     Spring::SpringType::BENDING));
        }
    }
}
```

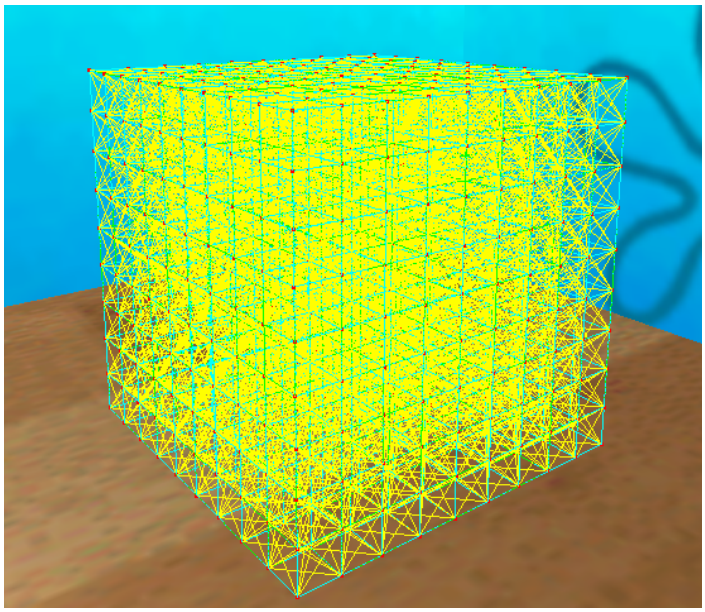
- SHEAR:

每個粒子的鄰近粒子扣掉前面算過的 STRUCT 所連過的 6 個方向,其餘剩下 26 – 6 = 20 種, 但因為 2 粒子間皆有 20 種方向, 因此只要實作 20 / 2 = 10 種方向就好。以下為示意的程式碼,其餘依此類推。

```
// shear
// direction - 下中前
for (int i = 0; i < particleNumPerEdge - 1; i++) {
    for (int j = 1; j < particleNumPerEdge; j++) {
        for (int k = 0; k < particleNumPerEdge; k++) {
            int iParticleID = i * particleNumPerFace + j * particleNumPerEdge + k;
            int iNeighborID = iParticleID + particleNumPerFace - particleNumPerEdge;
            Eigen::Vector3f SpringStartPos = particles[iParticleID].getPosition();
            Eigen::Vector3f SpringEndPos = particles[iNeighborID].getPosition();
            Eigen::Vector3f Length = SpringStartPos - SpringEndPos;
            float absLength = sqrt(Length[0] * Length[0] + Length[1] * Length[1] + Length[2] * Length[2]);

            springs.push_back(Spring(iParticleID, iNeighborID, absLength, springCoefStruct, damperCoefStruct,
                                     Spring::SpringType::SHEAR));
        }
    }
}
```

最後所連接完的 Jelly 長這樣:



- Internal force
  - Spring Force:
    遵守這個公式所算出 Spring Force

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

```
Eigen::Vector3f Jelly::computeSpringForce(const Eigen::Vector3f &positionA, const Eigen::Vector3f &positionB,
                                          const float springCoef, const float restLength) {
    // TODO#2-1: Compute spring force given the two positions of the spring.
    //   1. Review "particles.pptx"  from p.9 - p.13
    //   2. The sample below just set spring force to zero
    Eigen::Vector3f spring_force = Eigen::Vector3f::Zero();

    float deltax = (positionA - positionB).norm();
    spring_force = -springCoef * (deltax - restLength) * ((positionA - positionB) / deltax);

    return spring_force;
}
```

  - Damper Force
    遵守這個公式所算出 Spring Force

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

```
Eigen::Vector3f Jelly::computeDamperForce(const Eigen::Vector3f &positionA, const Eigen::Vector3f &positionB,
                                          const Eigen::Vector3f &velocityA, const Eigen::Vector3f &velocityB,
                                          const float damperCoef) {
    // TODO#2-2: Compute damper force given the two positions and the two velocities of the spring.
    //    1. Review "particles.pptx"  from p.9 - p.13
    //    2. The sample below just set damper force to zero

    Eigen::Vector3f damper_force = Eigen::Vector3f::Zero();

    Eigen::Vector3f deltav = velocityA - velocityB;
    Eigen::Vector3f deltax = positionA - positionB;
    damper_force = -damperCoef * (deltav.dot(deltax) / deltax.norm()) * (deltax / deltax.norm());

    return damper_force;
}
```

- Internal Force

歷遍每個 Spring 後,計算之間的 Spring Force 以及 Damper Force 再整合進粒子的總受力中。要注意的是除了一開始的粒子有受力,其結束端的粒子也有受到反方向的力。

```
for (const auto& spring: springs) {
    Eigen::Vector3f total_force;
    Eigen::Vector3f start_x = getParticle(spring.getSpringStartID()).getPosition();
    Eigen::Vector3f end_x = getParticle(spring.getSpringEndID()).getPosition();
    Eigen::Vector3f start_v = getParticle(spring.getSpringStartID()).getVelocity();
    Eigen::Vector3f end_v = getParticle(spring.getSpringEndID()).getVelocity();

    Eigen::Vector3f spring_force = computeSpringForce(start_x, end_x, spring.getSpringCoef(), spring.getSpringRestLength());
    Eigen::Vector3f damper_force = computeDamperForce(start_x, end_x, start_v, end_v, spring.getDamperCoef());

    total_force = spring_force + damper_force;

    getParticle(spring.getSpringStartID()).addForce(total_force);
    getParticle(spring.getSpringEndID()).addForce(-total_force);
}
```

- Collision

碰撞分為平面以及碗的部分。而其各粒子碰撞後的速度以及受力的公式如下:

速度:

$$V' = -k_r * V_N + V_T$$

受力分為

- Contact Force:

$$f^C = -(N \bullet f) * N \quad \text{for } N \bullet f < 0$$
$$f^C = 0 \quad \text{for } N \bullet f > 0$$

- Friction Force:

$$f^f = -k_f(-N \bullet f) * V_t \quad \text{for } N \bullet f < 0$$

- 碰撞偵測
  - Plant

平面的部分先偵測粒子距離平面上某點的 y 向量有沒有小於 eEPSIOL,以及粒子的速度和 normal 的內積有沒有小於 0 (檢查粒子是否是朝著平面

移動)，接著計算新的速度以及受力。需要注意的是要把洞的地方 continue 掉,不然 Jelly 不會掉進洞裡。

```
for (int i = 0; i < jelly.getParticleNum(); i++) {
    Particle &p = jelly.getParticle(i);
    Eigen::Vector3f pp = p.getPosition();
    // collision detect
    float ifonbowl = (pp - hole_position).norm();
    if (ifonbowl <= hole_radius) continue;

    float distance = (pp - position).dot(normal);
    if (distance < eEPSILON && p.getVelocity().dot(normal) < eEPSILON) {
        // velocity
        Eigen::Vector3f norV = p.getVelocity().dot(normal) * normal;
        Eigen::Vector3f tanV = p.getVelocity() - norV;
        Eigen::Vector3f newV = coefResist * norV + tanV;

        p.setVelocity(newV);

        // force
        Eigen::Vector3f conF = Eigen::Vector3f::Zero();
        Eigen::Vector3f friF = Eigen::Vector3f::Zero();
        Eigen::Vector3f totF;

        if (p.getForce().dot(normal) < 0) {
            conF = -(normal.dot(p.getForce())) * normal;
            friF = -coefFriction * (-normal.dot(p.getForce())) * tanV;
        }
        totF = conF + friF;
        p.addForce(totF);
    }

}
```

◆ Bowl

碗碰撞的偵測我使用了三種檢查的參數，分別是:

1. 半徑 - 粒子距離中心距離有沒有 ＜eEPSILON
2. 粒子的 y 向量有沒有小於 0 (平面下)
3. 粒子的速度和 normal 的內積有沒有小於 0

```
for (int i = 0; i < jelly.getParticleNum(); i++) {
    Particle& p = jelly.getParticle(i);
    Eigen::Vector3f pp = p.getPosition();
    float pmass = p.getMass();

    Eigen::Vector3f take_y_vec = Eigen::Vector3f(0.0f, 1.0f, 0.0f);
    float touch_ballwall = (pp - position).norm();
    Eigen::Vector3f normal = (position - pp).normalized();
    float underground = pp.dot(take_y_vec);

    if (underground <= 0 && abs(RADIUS - touch_ballwall) < eEPSILON && normal.dot(p.getVelocity()) < 0) {
        // velocity
        Eigen::Vector3f norV = p.getVelocity().dot(normal) * normal;
        Eigen::Vector3f tanV = p.getVelocity() - norV;
        Eigen::Vector3f newV = coefResist * (norV * (pmass - mass) / (pmass + mass)) + tanV;

        p.setVelocity(newV);

        // force
        Eigen::Vector3f conF = Eigen::Vector3f::Zero();
        Eigen::Vector3f friF = Eigen::Vector3f::Zero();
        Eigen::Vector3f totF;

        if (p.getForce().dot(normal) < 0) {
            conF = -(normal.dot(p.getForce())) * normal;
            friF = -coefFriction * (-normal.dot(p.getForce())) * tanV;
        }
        totF = conF + friF;

        p.addForce(totF);
    }
}
```

- Integration
  - **Explicit Euler Method**

    為每個粒子計算下一個時間點的位置和速度。

    ➔ 更新 = 原值 + deltaTime * 變化量

```
for (int i = 0; i < particleSystem.jellyCount; i++) {
    for (int j = 0; j < particleSystem.jellies[i].getParticleNum(); j++) {
        Particle &p = particleSystem.jellies[i].getParticle(j);
        p.setPosition(p.getPosition() + particleSystem.deltaTime * p.getVelocity());
        p.setVelocity(p.getVelocity() + particleSystem.deltaTime * p.getAcceleration());
        p.setForce(Eigen::Vector3f::Zero());
    }
}
```

  - **Implicit Euler Method**

    先把原始的資訊儲存起來，接著直接呼叫 computeJellyForce 再去計算一次力，因為做了碰撞偵測因此有些速度會改變，並以現在的速度以及加速度去更新初始狀態的位置以及速度。

```
std::vector<Particle> ori;

for (int i = 0; i < particleSystem.jellyCount; i++) {
    for (int j = 0; j < particleSystem.jellies[i].getParticleNum(); j++) {
        Particle tp;
        Particle& p = particleSystem.jellies[i].getParticle(j);
        tp.setPosition(p.getPosition());
        tp.setVelocity(p.getVelocity());
        ori.push_back(tp);

        p.setPosition(p.getPosition() + particleSystem.deltaTime * p.getVelocity());
        p.setVelocity(p.getVelocity() + particleSystem.deltaTime * p.getAcceleration());
        p.setForce(Eigen::Vector3f::Zero());
    }
}
particleSystem.computeJellyForce(particleSystem.jellies[0]);

for (int i = 0; i < particleSystem.jellyCount; i++) {
    for (int j = 0; j < particleSystem.jellies[i].getParticleNum(); j++) {
        Particle& p = particleSystem.jellies[i].getParticle(j);
        p.setVelocity(ori[j].getVelocity() + particleSystem.deltaTime * p.getAcceleration());
        p.setPosition(ori[j].getPosition() + particleSystem.deltaTime * ori[j].getVelocity());
    }
}
```

  - **Midpoint Euler Method**

    和 Implicit Euler method 很像，但是 Implicit Euler 是用 1 倍的時間差，而 Midpoint Euler 是使用 0.5 倍的時間差。

```
std::vector<Particle> ori;

for (int i = 0; i < particleSystem.jellyCount; i++) {
    for (int j = 0; j < particleSystem.jellies[i].getParticleNum(); j++) {
        Particle tp;
        Particle& p = particleSystem.jellies[i].getParticle(j);
        tp.setPosition(p.getPosition());
        tp.setVelocity(p.getVelocity());
        ori.push_back(tp);

        p.setPosition(p.getPosition() + (particleSystem.deltaTime * p.getVelocity()) / 2);
        p.setVelocity(p.getVelocity() + (particleSystem.deltaTime * p.getAcceleration()) / 2);
        p.setForce(Eigen::Vector3f::Zero());
    }
}
particleSystem.computeJellyForce(particleSystem.jellies[0]);

for (int i = 0; i < particleSystem.jellyCount; i++) {
    for (int j = 0; j < particleSystem.jellies[i].getParticleNum(); j++) {
        Particle& p = particleSystem.jellies[i].getParticle(j);
        p.setVelocity(ori[j].getVelocity() + particleSystem.deltaTime * p.getAcceleration());
        p.setPosition(ori[j].getPosition() + particleSystem.deltaTime * ori[j].getVelocity());
    }
}
```

■ **Runge Kutta 4th Method**

Runge Kutta 4^{th} method 需計算 k1, k2, k3, k4 。k1 是時間一開始的斜率、k2 是利用 k1 去求時間中點的斜率、k3 是利用 k2 所求的中點斜率、k4 為時間段結束的斜率,其 y 值由 k3 決定。 最後再依各項權重並用 Explicit Euler 計算最後的結果。

```
std::vector<StateStep> ori, k1, k2, k3, k4;
//k1
for (int i = 0; i < particleSystem.jellyCount; i++) {
    for (int j = 0; j < particleSystem.jellies[i].getParticleNum(); j++) {
        StateStep t, tk1;
        Particle& p = particleSystem.jellies[i].getParticle(j);

        t.vel = p.getVelocity();
        t.pos = p.getPosition();
        tk1.acc = particleSystem.deltaTime * p.getAcceleration();
        tk1.vel = particleSystem.deltaTime * p.getVelocity();
        p.setVelocity(p.getVelocity() + particleSystem.deltaTime * p.getAcceleration() / 2);
        p.setPosition( p.getPosition() + particleSystem.deltaTime * p.getVelocity() / 2);

        ori.push_back(t);
        k1.push_back(tk1);
    }
}
particleSystem.computeJellyForce(particleSystem.jellies[0]);

// k2
for (int i = 0; i < particleSystem.jellyCount; i++) {
    for (int j = 0; j < particleSystem.jellies[i].getParticleNum(); j++) {
        StateStep tk2;
        Particle& p = particleSystem.jellies[i].getParticle(j);

        tk2.acc = particleSystem.deltaTime * p.getAcceleration();
        tk2.vel = particleSystem.deltaTime * p.getVelocity();
        p.setVelocity(ori[j].vel + particleSystem.deltaTime * p.getAcceleration() / 2);
        p.setPosition(ori[j].pos + particleSystem.deltaTime * p.getVelocity() / 2);

        k2.push_back(tk2);
    }
}
particleSystem.computeJellyForce(particleSystem.jellies[0]);
```

```
// k3
for (int i = 0; i < particleSystem.jellyCount; i++) {
    for (int j = 0; j < particleSystem.jellies[i].getParticleNum(); j++) {
        StateStep tk3;
        Particle& p = particleSystem.jellies[i].getParticle(j);

        tk3.acc = particleSystem.deltaTime * p.getAcceleration();
        tk3.vel = particleSystem.deltaTime * p.getVelocity();
        p.setVelocity(ori[j].vel + particleSystem.deltaTime * p.getAcceleration());
        p.setPosition(ori[j].pos + particleSystem.deltaTime * p.getVelocity());

        k3.push_back(tk3);
    }
}
particleSystem.computeJellyForce(particleSystem.jellies[0]);

//k4
for (int i = 0; i < particleSystem.jellyCount; i++) {
    for (int j = 0; j < particleSystem.jellies[i].getParticleNum(); j++) {
        StateStep tk4;
        Particle& p = particleSystem.jellies[i].getParticle(j);

        tk4.acc = particleSystem.deltaTime * p.getAcceleration();
        tk4.vel = particleSystem.deltaTime * p.getVelocity();
        p.setVelocity(ori[j].vel + (k1[j].acc + 2 * k2[j].acc + 2 * k3[j].acc + tk4.acc) / 6);
        p.setPosition(ori[j].pos + (k1[j].vel + 2 * k2[j].vel + 2 * k3[j].vel + tk4.vel) / 6);

        k4.push_back(tk4);
    }
}
```

○ Result and Discussion

- The difference between integrators

  Runge Kutta Method 和 Midpoint Euler 可以看作是 Explicit Euler 中間值的應用。由於 Explicit Euler 是一步步計算的, Runge Kutta Method 是把一個時間段切成很多段, 再通過這些 k 值去計算最後一步。 而 Midpoint 就是只利用中間段的資訊。一般來說 Explicit Euler 與 Midpoint Euler 相比誤差較大, 而 Midpoint Euler 與 Runge Kutta 相比誤差較大。Implicit Euler method 則是要使用下一個時間段的資訊來求現在的資訊。通常我們都需要求解微分方程的根。

- Effect of parameters

  1. Spring coef & Damper coef:
     Spring coef 越大則相同受力下彈簧的形變量會越小, 而 Damper coef 越大使彈簧的震動越小。因此若是 Spring coef 太小且 Damper coef 太大會導致 unstable
  2. Resis coef:
     若是為非完全彈性碰撞的話會有能量消散, 因此會有一個恢復係數介於 0 到 1 之間, 越大能量消散越小。
  3. Friction coef:
     當物體和另一物體摩擦之間的係數, 越大則會導致摩擦力越大。

○ Bonus (Optional)

看到 Bonus 中有一項是 improve graphic, 雖然我不知道這算不算加分, 但是有看到 HW1 裡的材質包有一個海綿寶寶的造型,就給他換上了。

```
auto jelly1 = std::make_shared<gfx::Texture>(pf::find("Texture/spongebob_face.png"));
auto jelly2 = std::make_shared<gfx::Texture>(pf::find("Texture/spongebob_face.png"));
auto jelly3 = std::make_shared<gfx::Texture>(pf::find("Texture/spongebob_face.png"));
auto jelly4 = std::make_shared<gfx::Texture>(pf::find("Texture/spongebob_face.png"));
auto jelly5 = std::make_shared<gfx::Texture>(pf::find("Texture/spongebob_face.png"));
auto jelly6 = std::make_shared<gfx::Texture>(pf::find("Texture/spongebob_face.png"));
```

長這樣:

○ Conclusion

這次實作了一個蒟蒻的粒子系統, 其中模擬了 4 種 integrator 以及碰撞還有彈簧受力,以下是我的一些發現以及一個發現的 bug:

- 觀察 FPS 後發現，Integration method 為 Explicit Euler 時 FPS 較高, 而使用其他積分器時 FPS 較低。我認為應該是其他的 integrator 計算量較 Euler 大。但雖然照理來說其他 integrator 的精確度要較 Explicit Euler 高, 是肉眼無法明確識別精度,所以不能證明其他 integrator 的誤差比歐拉的小。
- 設定 parameter 時要小心, deltaT 不能太大也不能太小, Damper coef、Spring coef 都會影響系統的穩定性。像是我的 Implicit Euler 的 deltaTime 一定要小於 0.0007 才不會一開始就 unstable, 我認為有可能是我有某些地方計算錯誤之類的, 以至於若是用默認的 deltaTime 會導致受力暴增, 但是我真的 debug 很久都找不出來, 最後發現把 deltaTime 調小才可以跑。

最後我覺得實作一個物理引擎真的是很難, 且這也是我第一次寫那麼大型的專案, 每次 debug 都很久, 到一個新的 TODO 時也要先把相關的東西都先看過才能下手, 雖然最後 implicit Euler 有點小 bug, 但是其實還蠻有趣的。