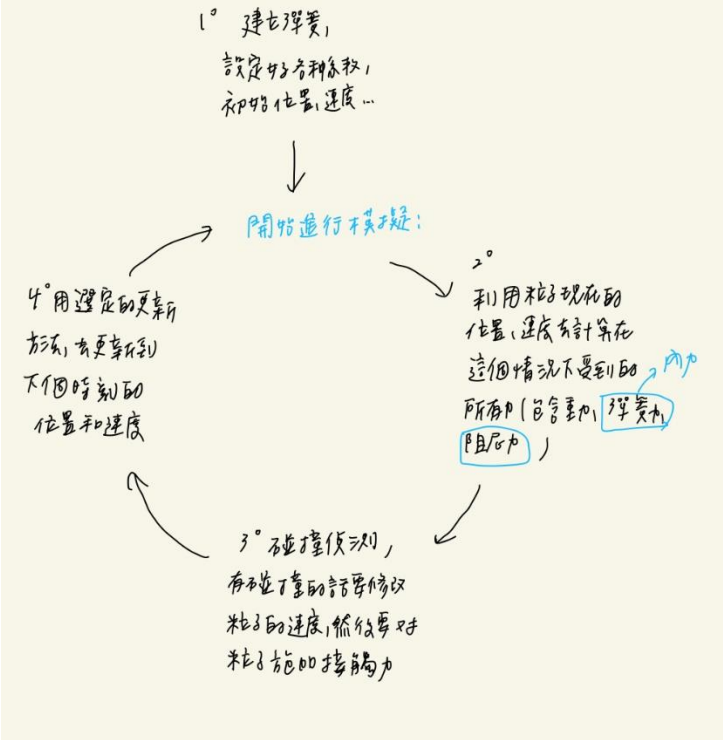
程式邏輯：



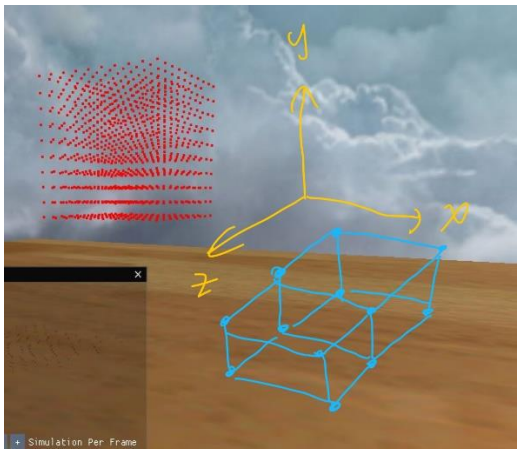
所以我以下的說明會像圖上的編號，先說明建立彈簧，再來說明計算內力、碰撞偵測，最後說明如何更新一步。

(1)

首先是連接彈簧的部分。

struct 彈簧是連接隔壁的彈簧，一個粒子直接的隔壁有上下左右前後共 6 個粒子，實際上在畫的時候不用每個粒子都連 6 個彈簧，因為彈簧有兩邊的關係，每個粒子實際上只要連三個就好，所以我分成 x, y, z 三個方向的彈簧來連。

Cube 的 x, y, z 方向示意圖:



首先是 x 方向的 struct 彈簧，我跑過每一個粒子，然後這個粒子的 id 就是彈簧的 iParticleID，而它隔壁的粒子 id 則是彈簧的 iNeighborID（這裡的隔壁粒子 id 是 iParticleID 再加上一整面的 particle 數量），然後彈簧的原始長度就是這兩個粒子間的長度，然後就和彈簧係數、damper 係數一起 construct 一個彈簧，然後 push 進這個 cube 中的 spring vector，就連好了一個彈簧。用相同的做法，就能連好其他 y, z 方向的彈簧（只是差別是 neighbor 的 id 會變成加每條 edge 的粒子數或是加 1）。連 bend 彈簧的方法也和連 struct 相同，我也是分成 x, y, z 三個方向去連，只是 bend 是隔了一個粒子連，所以 neighbor 的 id 加的數量就要變成加 2 倍。

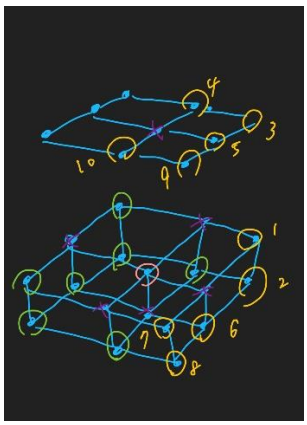
```
// z-direction
for (int i = 0; i < particleNumPerEdge; i++) {
    for (int j = 0; j < particleNumPerEdge; j++) {
        for (int k = 0; k < particleNumPerEdge - 1; k++) {

            int iParticleID = i * particleNumPerFace + j * particleNumPerEdge + k;
            int iNeighborID = iParticleID + 1;
            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));


        }
    }
}
```

再來是比較麻煩的 shear 彈簧，是斜著連，一個粒子總共要連 10 條彈簧出去（但如果粉紅粒子是在 cube 邊緣的話有可能就連不到 10 條）。對於在中心的粉紅粒子而言，它所要連的 10 條彈簧就是我在下圖用黃筆圈起來的 1～10 號粒子，綠色的部分因為彈簧對稱，所以是那個粒子會來連到這個粉紅粒子。實作時我是一次把每個粒子的 10 條邊連好，用中心粒子的 x, y, z 來判斷它有哪些粒子可以連，然後像 struct, bend 彈簧一樣計算 neighbor 的 id（這裡的 neighbor id 的算法都不一樣，不過基本上就是拿中心粒子 id 去加減每面、每邊的粒子數，然後因為是斜的，所以可能會再加減 1）、原始長度等，然後 push 進 spring vector，把彈簧連好。



(2)

再來是計算彈簧力和阻尼力的部分。

彈簧力的話，利用彈簧兩端的粒子的位置減掉彈簧的原始長度就能獲得彈簧伸長量，再把彈簧兩端粒子位置相減的單位向量作為彈簧力的方向向量，最後把伸長量、方向向量乘上彈簧常數再乘上負號就能得到彈簧力了。

```
Eigen::Vector3f Cube::computeSpringForce(const Eigen::Vector3f &positionA, const Eigen::Vector3f &positionB,
                                         const float springCoef, const float restLength) {
    // TODO
    Eigen::Vector3f distance_vector = positionA - positionB;
    float distance = sqrt(distance_vector[0] * distance_vector[0] + distance_vector[1] * distance_vector[1] +
                          distance_vector[2] * distance_vector[2]);

    float deltaLength = distance - restLength;
    Eigen::Vector3f spring_force = -springCoef * deltaLength * distance_vector / distance;
    return spring_force;
}
```

接著是阻尼力，阻尼力像是速度的彈簧，所以在計算上，除了會用到彈簧兩端粒子的位置，也會用到它們的速度。這裡一樣把彈簧兩端粒子位置相減的單位向量作為阻尼力的方向向量，然後剛才的彈簧力用的是彈簧伸長量，但這裡因為是速度的彈簧，所以要用的是彈簧兩端粒子的速度差（相對速度），且要把這個相對速度投射到方向向量上的值才是真正會影響阻尼力的值，最後也是把它們乘上阻尼係數和負號就能得到阻尼力了。

```
Eigen::Vector3f Cube::computeDamperForce(const Eigen::Vector3f &positionA, const Eigen::Vector3f &positionB,
                                         const Eigen::Vector3f &velocityA, const Eigen::Vector3f &velocityB,
                                         const float damperCoef) {
    // TODO

    Eigen::Vector3f distance_vector = positionA - positionB;
    float distance = sqrt(distance_vector[0] * distance_vector[0] + distance_vector[1] * distance_vector[1] +
                          distance_vector[2] * distance_vector[2]);
    Eigen::Vector3f relative_velocity = velocityA - velocityB;
    Eigen::Vector3f damper_force =
        -damperCoef * relative_velocity.dot(distance_vector) / distance * distance_vector / distance;
    return damper_force;
}
```

算好彈簧力和阻尼力後，它們會相加起來成為粒子的內力，然後被施加在彈簧兩端的粒子上。因為彈簧兩端的粒子都要施加力，而我在計算彈簧力和阻尼力時使用的方向向量是由彈簧的末端粒子指向彈簧的起始粒子，所以我所計算的彈簧力和阻尼力都是對起始粒子的，所以在施加力到末端粒子時要乘負號。

```
void Cube::computeInternalForce() {
    // TODO
    for (int i = 0; i < springs.size(); i++) {

        int A_id = springs[i].getSpringStartID();
        int B_id = springs[i].getSpringEndID();
        Eigen::Vector3f A_position = particles[A_id].getPosition();
        Eigen::Vector3f B_position = particles[B_id].getPosition();
        Eigen::Vector3f A_velocity = particles[A_id].getVelocity();
        Eigen::Vector3f B_velocity = particles[B_id].getVelocity();

        Eigen::Vector3f spring_force =
            computeSpringForce(A_position, B_position, springs[i].getSpringCoef(), springs[i].getSpringRestLength());
        Eigen::Vector3f damper_force =
            computeDamperForce(A_position, B_position, A_velocity, B_velocity, springs[i].getDamperCoef());

        Eigen::Vector3f net_force = spring_force + damper_force;

        particles[A_id].addForce(net_force);
        particles[B_id].addForce(-net_force);
```

(3)

再來是碰撞偵測的部分。

首先是平地，在這裡會看過每一顆粒子，我用粒子的位置減掉平面的位置作為相對位置，然後如果這個相對位置和平面法向量（已是單位向量）的內積（相當於是粒子和平面的距離）小於 epsion 而且粒子的速度和平面法向量內積小於 0（相當於粒子現在正往平面飛去）的話，就表示碰撞發生。所以我計算粒子的法線速度（平面法向量內積粒子速度會得到負的法線速度的大小，再乘上法向量後就能變成正的法線速度），再用粒子速度減掉法線速度拿到切線速度。此時因為碰撞發生，所以要修改粒子的速度，其新的法線速度是原來的法線速度乘上負的衰減係數（負號表示方向相反，因為碰撞了會反彈），新的切線速度則和原來一樣。然後還要判斷是否要對粒子施加接觸力，如果粒子本來的力去內積平面法向量小於 0 的話（表示粒子受到往平面施加的力），那就要對粒子施加額外的接觸力，來抵消這個往平面的力，這個用來抵消的力必需和粒子原本法向量方向的力大小相同方向相反（所以把粒子的力內積平面法向量，再乘上平面法向量，得到粒子法向量方向的力，再加負號）。除了這個抵消的力，也需要有摩擦力，而摩擦力就是負的切線速度方向（這個負號會在平面法向量內積粒子的力得到）、大小是正向力乘上摩擦係數，然後就把算出來這兩個力再施加給粒子）。

而斜面的部分，和平面的做法幾乎一模一樣，唯一的差別只在斜面的法向量給的不是單位向量，所以我先把它變成單位向量，之後所有要和法向量內積都是和這個變成單位向量的法向量內積。

```cpp
void PlaneTerrain::handleCollision(const float delta_T, Cube& cube) {
    constexpr float eEPSILON = 0.01f;
    constexpr float coefResist = 0.8f;
    constexpr float coefFriction = 0.3f;
    // TODO
    for (int i = 0; i < cube.getParticleNum(); i++) {
        Particle& particle = cube.getParticle(i);

        Eigen::Vector3f relative_position = particle.getPosition() - position;

        // if the particle and the wall is close enough and if the particle is moving forward to the wall
        // collision happens
        if (normal.dot(relative_position) < eEPSILON && normal.dot(particle.getVelocity()) < 0) {

            Eigen::Vector3f normal_velocity = normal.dot(particle.getVelocity()) * normal;
            Eigen::Vector3f tangent_velocity = particle.getVelocity() - normal_velocity;

            particle.setVelocity(-coefResist * normal_velocity + tangent_velocity);

            // if a force pushes the particle into the wall
            // exert a contact force (resist + friction) to resist it
            if (normal.dot(particle.getForce()) < 0) {

                Eigen::Vector3f resist = -normal.dot(particle.getForce()) * normal;
                Eigen::Vector3f friction =
                    coefFriction * normal.dot(particle.getForce()) * tangent_velocity / tangent_velocity.norm();
                particle.addForce(resist + friction);
```

```cpp
void TiltedPlaneTerrain::handleCollision(const float delta_T, Cube& cube) {
    constexpr float eEPSILON = 0.01f;
    constexpr float coefResist = 0.8f;
    constexpr float coefFriction = 0.3f;
    // TODO

    Eigen::Vector3f unit_normal = normal / normal.norm();

    for (int i = 0; i < cube.getParticleNum(); i++) {
        Particle& particle = cube.getParticle(i);

        Eigen::Vector3f relative_position = particle.getPosition() - position;

        // if the particle and the wall is close enough and if the particle is moving forward to the wall
        // collision happens
        if (unit_normal.dot(relative_position) < eEPSILON && unit_normal.dot(particle.getVelocity()) < 0) {
            Eigen::Vector3f normal_velocity = unit_normal.dot(particle.getVelocity()) * unit_normal;
            Eigen::Vector3f tangent_velocity = particle.getVelocity() - normal_velocity;

            particle.setVelocity(-coefResist * normal_velocity + tangent_velocity);

            // if a force pushes the particle into the wall
            // exert a contact force (resist + friction) to resist it
            if (unit_normal.dot(particle.getForce()) < 0) {
                Eigen::Vector3f resist = -unit_normal.dot(particle.getForce()) * unit_normal;
                Eigen::Vector3f friction =
                    coefFriction * unit_normal.dot(particle.getForce()) * tangent_velocity / tangent_velocity.norm();
                particle.addForce(resist + friction);
```

然後球的部份，我先計算粒子和球心的相對位置然後把它單位化，用來作為粒子對球面的法向量（是出球面方向）。然後如果粒子到球心的距離檢定球的半徑小於 epison（表示粒子離球面很近或是穿到球裡了）且相對位置內積粒子速度小於 0（表示粒子正往球飛），碰撞發生。和平面時一樣計算法線速度、切線速度，不過這裡內積的法向量都是我剛才算出來的單位化的相對位置，然後新的速度則是利用 spec 中給的公式（粒子為 v1，球面為 v2，粒子質量 m1，球面質量 m2，球面不動速度為 0）算出碰撞後的新速度（不過在法線方向要乘上衰減係數，而因為 m1-m2 小於 0，所以新的法線速度會是反向）。然後一樣判斷是否要施加接觸力給粒子（如果法向量內積粒子的力小於 0，表示粒子有向球面施力，就要接觸力），抵消法向量方向的力一樣是用法向量內積粒子的力再乘上負的法向量，摩擦力也一樣是負的切線速度方向、大小是正向力乘上摩擦係數。

```cpp
void SphereTerrain::handleCollision(const float delta_T, Cube& cube) {
    constexpr float eEPSILON = 0.01f;
    constexpr float coefResist = 0.8f;
    constexpr float coefFriction = 0.3f;
    // TODO

    for (int i = 0; i < cube.getParticleNum(); i++) {
        Particle& particle = cube.getParticle(i);

        Eigen::Vector3f relative_position = particle.getPosition() - position;
        Eigen::Vector3f unit = relative_position / relative_position.norm();

        // if the particle and the sphere is close enough and if the particle is moving forward to the sphere
        // collision happens
        if (relative_position.norm() - radius < eEPSILON && relative_position.dot(particle.getVelocity()) < 0) {
            Eigen::Vector3f normal_velocity = unit.dot(particle.getVelocity()) * unit;
            Eigen::Vector3f tangent_velocity = particle.getVelocity() - normal_velocity;

            particle.setVelocity(coefResist * normal_velocity * (particle.getMass() - mass) /
                                 (particle.getMass() + mass) +
                             tangent_velocity);

            // if a force pushes the particle into the sphere
            // exert a contact force (resist + friction) to resist it
            if (unit.dot(particle.getForce()) < 0) {
                Eigen::Vector3f resist = -unit.dot(particle.getForce()) * unit;
                Eigen::Vector3f friction =
                    coefFriction * unit.dot(particle.getForce()) * tangent_velocity / tangent_velocity.norm();
                particle.addForce(resist + friction);
            }
```

碗的話和球一樣是利用粒子和碗心的相對位置然後把它單位化，用來作為粒子對碗面的法向量（是出碗面方向）。然後在判斷碰撞是否發生時和球相反，變成是用半徑減掉相對位置，如果值小於 epison 且相對位置內積粒子速度大於 0 就表示有碰撞。然後和球一樣計算法線速度和切線速度，然後計算碰撞後的新速度。接著判斷是否要施加接觸力，和球相反，這裡要法向量內積粒子的力大於 0 才表示粒子有向碗施力，然後計算抵消法線方向的力，也計算摩擦力（這裡因為法向量內積粒子的力會是正的，所以要加負號作為負的切線方向的「負」）。

```cpp
void BowlTerrain::handleCollision(const float delta_T, Cube& cube) {
    constexpr float eEPSILON = 0.01f;
    constexpr float coefResist = 0.8f;
    constexpr float coefFriction = 0.3f;
    // TODO

    for (int i = 0; i < cube.getParticleNum(); i++) {
        Particle& particle = cube.getParticle(i);

        Eigen::Vector3f relative_position = particle.getPosition() - position;
        Eigen::Vector3f unit = relative_position / relative_position.norm();

        // if the particle and the bowl is close enough and if the particle is moving forward to the bowl
        // collision happens
        if (radius - relative_position.norm() < eEPSILON && relative_position.dot(particle.getVelocity()) > 0) {
            Eigen::Vector3f normal_velocity = unit.dot(particle.getVelocity()) * unit;
            Eigen::Vector3f tangent_velocity = particle.getVelocity() - normal_velocity;


            particle.setVelocity(coefResist * normal_velocity * (particle.getMass() - mass) /
                                 (particle.getMass() + mass) +
                             tangent_velocity);

            // if a force pushes the particle into the bowl
            // exert a contact force (resist + friction) to resist it
            if (unit.dot(particle.getForce()) > 0) {
                Eigen::Vector3f resist = - unit.dot(particle.getForce()) * unit;
                Eigen::Vector3f friction =
                    -coefFriction * unit.dot(particle.getForce()) * tangent_velocity / tangent_velocity.norm();
                particle.addForce(resist + friction);
```

(4)

然後是更新每一步的方法。

首先是顯尤拉法，顯尤拉法是利用現在的速度及加速度來更新下一個時刻的位置及速度。所以我在顯尤拉裡面做的事，就是看過每一個粒子，然後把它們的位置加上 delta t 乘上粒子速度變成下一刻的位置，然後把粒子速度加上 delta t 乘上粒子加速度變成下一刻的速度。然後把粒子受到的力清成 0，繼續下一個時刻的模擬。

```
void ExplicitEulerIntegrator::integrate(MassSpringSystem& particleSystem) {
    // TODO
    // float deltaTime = particleSystem.deltaTime;
    for (int i = 0; i < particleSystem.getCubeCount(); i++) {
        Cube* cube = particleSystem.getCubePointer(i);

        for (int j = 0; j < cube->getParticleNum(); j++) {
            Particle& particle = cube->getParticle(j);

            // x=x0+vt
            particle.addPosition(particleSystem.deltaTime * particle.getVelocity());

            Eigen::Vector3f acceleration = particle.getAcceleration();
            // v=v0+at
            particle.addVelocity(particleSystem.deltaTime * acceleration);

            // clear force
            particle.setForce(Eigen::Vector3f::Zero());
```

接著是中點法，中點法是利用中點（也就是用現在的速度更新半步的地方）的速度及加速度來更新下一個時刻的位置及速度。所以我在中點法裡面做的事，就是看過每一個粒子，先把它們的現在的位置和速度存起來，然後再把粒子位置加上 0.5*delta t 乘上粒子現在速度變成中點的位置，然後把粒子速度加上 0.5*delta t 乘上粒子現在的加速度變成中點的速度，先把粒子的力清乾淨，然後呼叫 computeCubeForce 這個 function，讓它在這個中點的狀態下，去計算粒子會受到多少力，來獲得中點的加速度，接著把剛才存起來的原始位置和速度加上 delta t 乘上現在得到的中點速度和加速度就完成了一步的更新了。然後一樣把粒子受到的力清成 0，繼續下一個時刻的模擬。

```
for (int i = 0; i < particleSystem.getCubeCount(); i++) {
    Cube* cube = particleSystem.getCubePointer(i);

    std::vector<Eigen::Vector3f> current_position, current_velocity;

    for (int j = 0; j < cube->getParticleNum(); j++) {
        Particle& particle = cube->getParticle(j);

        // retain the current position and velocity
        current_position.push_back(particle.getPosition());
        current_velocity.push_back(particle.getVelocity());

        // change the position to mid point position
        // change velocity to mid point velocity
        particle.addPosition(0.5 * particleSystem.deltaTime * particle.getVelocity());
        particle.addVelocity(0.5 * particleSystem.deltaTime * particle.getAcceleration());

        // clear force
        particle.setForce(Eigen::Vector3f::Zero());
    }

    // calculate the mid point force
    particleSystem.computeCubeForce(*cube);

    // mid point information
    for (int j = 0; j < cube->getParticleNum(); j++) {
        Particle& particle = cube->getParticle(j);

        particle.setPosition(current_position[j] + particleSystem.deltaTime * particle.getVelocity());
        particle.setVelocity(current_velocity[j] + particleSystem.deltaTime * particle.getAcceleration());

        // clear force
```

Runge kutta 的話，是利用四個不同時刻和位置的速度和加速度做權重平均來更新粒子下一刻的位置和速度。在這裡我一樣看過每一顆粒子，先把它們的現在的位置和速度存起來，然後也記錄用現在的速度和加速度更新一步的位移量和速度變化量。然後再把粒子位置加上 0.5*delta t 乘上粒子現在速度變成中點的位置，然後把粒子速度加上 0.5*delta t 乘上粒子現在的加速度變成中點的速度，一樣把粒子的力清乾淨，然後呼叫 computeCubeForce 這個 function，讓它在這個中點的狀態下，去計算粒子會受到多少力，來獲得中點的加速度，然後也是記錄用這個中點速度和加速度更新一步的位移量和速度變化量，接著把剛才存起來的原始位置和速度加上 0.5*delta t 乘上現在得到的中點速度和加速度再變成下一個狀態，一樣計算此狀態的受力、加速度，把用這個狀態速度、加速度更新一步的位移量和速度變化量存起來，再更新到第四個狀態（利用第三個狀態的速度和加速度更新一步的地方）一樣算力、把位移量和速度變化量存起來。接著利用存起來的這四個位移量和速度變化量作權重平均，然後加到一開始存起來的原始位置和速度，就完成了用 runge kutta 做的一步更新。

最後是隱尤拉法，隱尤拉法的想法是利用下一刻的速度及加速度（用現在的速度和加速度更新一步的地方）來更新下一個時刻的位置及速度。這麼做的實現方法，會變得像中點法一樣，只是在中點法乘的 0.5*delta t 在這裡會變成乘 delta t 去變成下一刻的狀態。雖然這樣子的做法在想法上是沒錯的，但實際上如果真的這麼做的話，會使得誤差變得太大（因為拿了比中點法還遠的地方的資訊來更新現在的位置和速度），所以在實現隱尤拉時，會先把現在的位置和速度存起來，然後把力清掉，就接在這個情況下再去算一次力（因為前面做了

碰撞偵測，有些粒子的速度是改變的，所以連續做兩次的結果並不會一樣），就以現在算出來的速度和加速度代表下一刻的速度和加速度，然後乘上 delta t 之後加上存起來的原始位置和速度，就完成隱尤拉的一步更新。

```
void ImplicitEulerIntegrator::integrate(MassSpringSystem& particleSystem) {
    // TODO

    for (int i = 0; i < particleSystem.getCubeCount(); i++) {
        Cube* cube = particleSystem.getCubePointer(i);

        std::vector<Eigen::Vector3f> current_position, current_velocity;

        for (int j = 0; j < cube->getParticleNum(); j++) {
            Particle& particle = cube->getParticle(j);

            // retain the current position and velocity
            current_position.push_back(particle.getPosition());
            current_velocity.push_back(particle.getVelocity());

            // clear force
            particle.setForce(Eigen::Vector3f::Zero());
        }

        // calculate the next point force
        particleSystem.computeCubeForce(*cube);

        // next point information
        for (int j = 0; j < cube->getParticleNum(); j++) {
            Particle& particle = cube->getParticle(j);

            particle.setPosition(current_position[j] + particleSystem.deltaTime * particle.getVelocity());
            particle.setVelocity(current_velocity[j] + particleSystem.deltaTime * particle.getAcceleration());

            // clear force
```

(5)

一些比較及討論

我覺得幾種更新方法中，能看得出最明顯不同的地方應該是在斜面的情況下。隱尤拉在從斜面滑下去的時候，cube 只旋轉很少很少，但是其他三個都能看得出有明顯的旋轉，可能是因為隱尤拉是直接再算一次力更新速度，所以和其他三個比起來，兩步之間的差距沒有那麼大，所以才有這樣的結果。

然後是不同的參數設定，首先我試著把彈簧常數調高，發現在平面的時候，cube 彈到地上反彈的時候彈的比較高，很符合彈簧常數高的彈簧比較彈的感覺。同樣的效果，也可以透過減少阻尼係數來得到（也可以透過把掉落的 y 座標增加得到），原因應該是因為阻尼係數變小，系統內部的摩擦就變小，就有比較多力可以繼續彈下去的感覺。不過雖然兩者呈現出來的效果很類似，但在係數的增減量卻差很多，我把彈簧常數多加了大約 3000 的結果，在阻尼係數卻只要減少 20 就能有差不多的效果。

然後我還試了讓掉下來的時候，會有一點斜著掉下來，雖然剛碰到地面的時候，它會有點歪歪的彈，但到最後它會越來越平衡然後回正。