

Computer Animation & Special Effects: HW1 Report

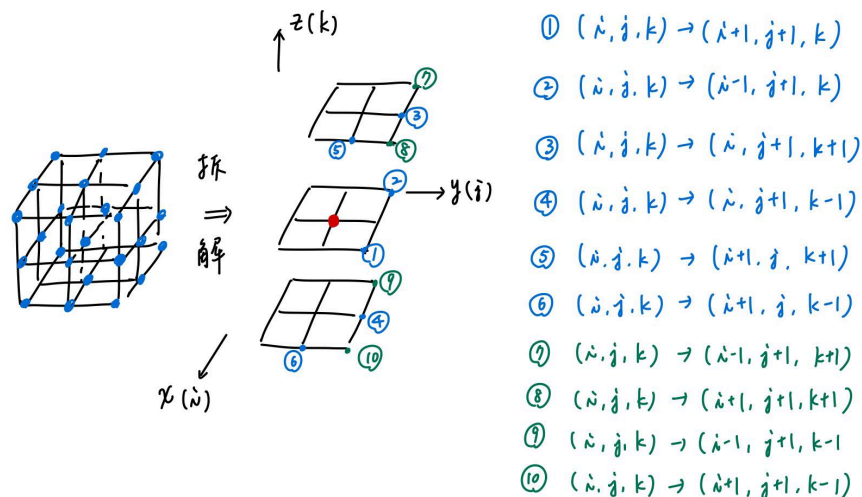
Author: 111550113 謝詠晴

1 Implementation

1.1 Jelly

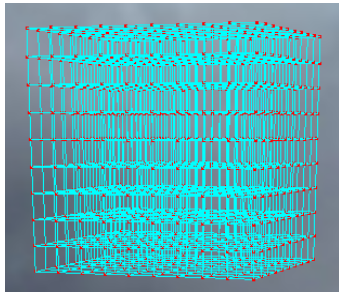
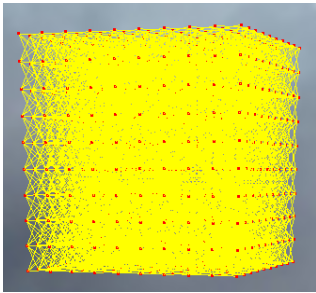
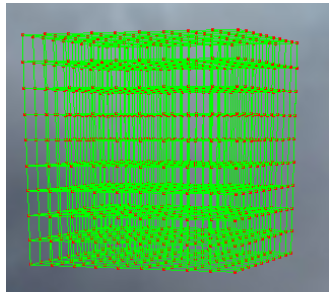
首先在 initializeSpring() 要將粒子用三種類型的彈簧連接起來，模擬具有彈性與可變形特性的果凍物體。三種彈簧分別為：Struct, Shear, Bending，以下將分別解釋。

- (1) Struct 彈簧：和隔壁粒子連接，一個粒子總共會有六個方向（上下左右前後），但因為彈簧有兩個端點，因此實際上只要畫三個方向(x, y, z) 即可。
- (2) Shear 彈簧：要連接斜向的粒子，具體的十個斜向方向可參考下圖我繪製的示意圖。而這部分在實作時要注意 neighbor id 的 i, j, k 因為會有加減1的問題，有時須改成從 1 開始或到 particleNumPerEdge - 1。



- (3) Bending 彈簧：和 Struct 彈簧的構建方法類似，差別在於會是和間隔一個鄰居的粒子連接，因此在尋找 neighbor id 時都會要加 2 倍。

每一條彈簧都會被加入一個彈簧列表中，並會記錄下構建該彈簧的起點和終點的質點編號、靜止長度、彈性係數與阻尼係數，以及該彈簧的類型。

		
Struct	Shear	Bending

接著在computeSpringForce() 和 computeDamperForce() 依照下圖公式計算彈簧力和阻尼力。最後由computeInternalForce() 將這兩力加總為一個總內部力，並對於每一條彈簧的兩端粒子施加此力，其中終點端粒子受到的力會是反向的。

$$\text{spring force} = -k_s(|x_a - x_b| - l_0) \cdot \frac{x_a - x_b}{|x_a - x_b|}$$

$$\text{damper force} = -k_d \left[\frac{(v_a - v_b)(x_a - x_b)}{|x_a - x_b|} \right] \frac{x_a - x_b}{|x_a - x_b|}$$

1.2 handleCollision: Plane Terrain

- (1) vecN 是平面的法向量，要達成碰撞偵測有以下兩個條件：
 - (a) 當 vecN 與粒子到平面的距離的向量內積小於 epsilon，表示該粒子夠靠近平面。
 - (b) 當 vecN 和粒子速度內積小於 0，表示粒子是朝向平面前進的
- (2) 達成這兩個條件表示會造成碰撞，接著要更新粒子在碰撞後的速度。我將速度拆分成法線方向的速度和切線方向的速度。其中切線方向速度不變，而法線方向速度則要多乘上恢復係數，並且是負值(因為反彈是反方向)。
- (3) 下一步則要判斷是否要對粒子施加接觸力和摩擦力，條件是當vecN和粒子本身的力內積小於 0，表示施力方向朝牆面，因此要對粒子施加接觸力和摩擦力來抵銷此力。
 - (a) 接觸力，反向於法線方向，大小為粒子原本總力在法線方向的投影值。
 - (b) 摩擦力，方向為切線方向，大小為摩擦係數乘以接觸力大小。

```
Eigen::Vector3f vecN = this->normal.normalized();

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

    Eigen::Vector3f particlePos = particle.getPosition();
    Eigen::Vector3f particleVel = particle.getVelocity();
    Eigen::Vector3f particleForce = particle.getForce();

    if(vecN.dot(particlePos - this->position) < eEPSILON && vecN.dot(particleVel) < 0) {
        // Collision happens
        // 1. Directly update particles' velocity
        Eigen::Vector3f velocityN = particleVel.dot(vecN) * vecN;
        Eigen::Vector3f velocityT = particleVel - velocityN;
        Eigen::Vector3f newVel = -coefResist * velocityN + velocityT;
        particle.setVelocity(newVel);

        // 2. Apply contact force to particles when needed
        if(vecN.dot(particleForce) < 0) {
            Eigen::Vector3f contactForce = -(vecN.dot(particleForce) * vecN);
            particle.addForce(contactForce);

            Eigen::Vector3f frictionForce = -coefFriction * (-vecN.dot(particleForce)) * velocityT;
            particle.addForce(frictionForce);
        }
    }
}
```

1.3 handleCollision: Elevator Terrain

- (1) VecN 是電梯平面的法向量，而進行碰撞偵測要滿足的兩個條件和 Plane 一樣：
 - (a) 當 vecN 與粒子到平面的距離的向量內積小於 epsilon，表示該粒子夠靠近電梯表面。
 - (b) vecN 和粒子與電梯的相對速度內積小於 0，表示粒子是朝向電梯平面運動的。
- (2) 若滿足碰撞條件表示發生碰撞，需要更新粒子速度，一樣將速度分為法線和切線方向，而切線方向不變。法線方向的速度則經過彈性碰撞公式： $new_u_a = (m1*u1 + m2*u2 + m2*coefResist*(u2 - u1)) / (m1 + m2)$ 的處理。u1 是粒子的法線速度分量，u2 是電梯的法線速度分量，m1 是粒子質量，m2 是電梯質量。
- (3) 最後一樣判斷是否要對粒子施加接觸力和摩擦力，這部分的實作和 plane 相同。

```
Eigen::Vector3f vecN = this->normal.normalized();

for (int i = 0; i < jelly.getParticleNum(); i++) {
    Particle& particle = jelly.getParticle(i);
    float particleMass = particle.getMass();
    Eigen::Vector3f particlePos = particle.getPosition();
    Eigen::Vector3f particleVel = particle.getVelocity();
    Eigen::Vector3f relativeVel = particleVel - this->velocity;

    if(vecN.dot(particlePos - this->position) < eEPSILON && vecN.dot(relativeVel) < 0) {
        // 1. Directly update particles' velocity
        Eigen::Vector3f velocityN = particleVel.dot(vecN) * vecN;
        Eigen::Vector3f velocityT = particleVel - velocityN;

        float u_a = vecN.dot(particleVel);
        float u_b = vecN.dot(this->velocity);
        float new_u_a = (particleMass * u_a + this->mass * u_b + this->mass * coefResist * (u_b - u_a)) / (particleMass + this->mass);
        Eigen::Vector3f newVel = new_u_a * normal + velocityT;
        particle.setVelocity(newVel);

        // 2. Apply contact force to particles when needed
        if(vecN.dot(particle.getForce()) < 0) {
            Eigen::Vector3f contactForce = -(vecN.dot(particle.getForce()) * vecN);
            particle.addForce(contactForce);

            Eigen::Vector3f frictionForce = -coefFriction * (-vecN.dot(particle.getForce())) * velocityT;
            particle.addForce(frictionForce);
        }
    }
}
```

2 Result and Discussion

2.1 The difference between integrators

Integrators 用來根據微分方程計算粒子運動(如位置與速度)隨時間的變化，主要依據 delta time 來近似系統狀態在下一個時間點的值。

- (1) Explicit Euler
 - (a) 計算方式：直接使用目前的速度與加速度預測下一時刻的位置與速度。
 - (b) 公式：
 - (i) $X(n+1) = X(n) + V(n) * dt$
 - (ii) $V(n+1) = V(n) + A(n) * dt$

(c) 優點:簡單容易實作、效率高

(d) 缺點:準確度低, 不穩定, 當 delta time 太大或系統變化劇烈, 容易失控

(2) Implicit Euler

(a) 計算方式:使用預測的下一時刻的速度和加速度來得到下一個時刻的位置與速度。首先暫存備份目前狀態的數值, 再來清空舊的力, 並使用新位置下的狀態重新計算出新的力 $F(n+1)$ 。接著更新粒子與電梯的速度和位置。

(b) 公式:

$$(i) \quad X(n+1) = X(n) + V(n+1) * dt$$

$$(ii) \quad V(n+1) = V(n) + A(n+1) * dt$$

(c) 優點:穩定性高, 適合剛性系統

(d) 缺點:計算成本高、耗時

(3) Midpoint Euler

(a) 計算方式:先計算中間時間點下的速度與位置, 再用該中間值來估算下一個時間點的位置與速度。首先計算了 midpos 和 midvel 並將其暫存到 particle 的狀態, 接著用 midpoint 的速度位置重新計算力, 得到 $A(mid)$ 。下一步即根據 midpoint 結果進行真正的更新。

(b) 公式:

$$(i) \quad V(mid) = V(n) + \frac{1}{2} * A(n) * dt$$

$$(ii) \quad X(mid) = X(n) + \frac{1}{2} * V(n) * dt$$

$$(iii) \quad A(mid) = F(Xmid, Vmid) / m$$

$$(iv) \quad X(n+1) = X(n) + V(mid) * dt$$

$$(v) \quad V(n+1) = V(n) + A(mid) * dt$$

(c) 優點:計算量適中, 準確度優於 Explicit Euler

(d) 缺點:還是可能不穩定, 誤差比 Runge-Kutta 高

(4) Runge-Kutta Fourth

(a) 計算方式:透過四次不同時刻的速度和位置的預估來做權重平均速度變化

(b) 公式:

$$(i) \quad k1 = f(x0, t0) * dt$$

$$(ii) \quad k2 = f(x0 + 0.5 * k1, t0 + 0.5 * dt) * dt$$

$$(iii) \quad k3 = f(x0 + 0.5 * k2, t0 + 0.5 * dt) * dt$$

$$(iv) \quad k4 = f(x0 + k3, t0 + dt) * dt$$

$$(v) \quad X(t0 + dt) = X(t0) + (k1 + 2k2 + 2k3 + k4) / 6$$

(c) 優點:使用較大的 delta time 仍可以有高準確度且穩定

(d) 缺點:每一步需要計算四次力, 且暫存中間狀態需要額外記憶體

2.2 Effect of parameters (springCoef, damperCoef, coefResist, coefFriction, etc.)

(1) springCoef:

- (a) 數值越大, 表示彈簧的回復力越強, 果凍的震動效果越明顯、越有彈性。
 - (b) 但當數值設過大時, 若使用不穩定的 Integrators 如 Explicit Euler 可能會造成爆震的情況。
- (2) damperCoef:
- (a) 阻尼係數越大, 表示果凍整體的阻力越大, 震動會更快速減弱, 使得回彈的幅度會變小。
 - (b) 當數值設為 0 時, 果凍會整個縮起來而卡住。
- (3) coefResist:
- (a) 控制果凍與接觸平面碰撞後的反彈程度。接近 1 時會像是完全碰撞, 很明顯的反彈回去, 而設成 0 則幾乎就不會反彈。
- (4) coefFriction:
- (a) 決定接觸面間的摩擦係數, 數值越大表示摩擦越大, 果凍會越難滑動。而當數值設為 0, 表示沒有靜摩擦力, 果凍會直接滑走。

3 Conclusion

經過我實作和觀察不同種 integrators 的效果後得到了以下的結論:

- (1) 四種方法在一般參數設定下, 模擬結果從外觀上來看差異不大。但在運算速度上可以感受到區別:
 - (a) Runge-Kutta 是最準確的方法, 但在實作上明顯比其他方法慢許多, 特別是在粒子數多的情況下更為明顯。
 - (b) Explicit Euler 計算效率最高, 運動順暢, 反應速度快。
- (2) 在 slope terrain 的特殊狀況:
 - (a) 僅有 Implicit Euler 的結果會讓 jelly 滾一圈後就滑下坡面, 其他方法則會出現果凍在坡面上持續翻滾的現象。
- (3) 極端參數測試結果:
 - (a) 當設定 springCoef = 2400、damperCoef = 120 時, 只有 Runge-Kutta 不會出現爆震或發散現象, 其他三種方法皆會導致模擬不穩定, 位置與速度均迅速發散。
 - (b) 當將 delta time 設為較大的值(例如 0.0027)時, 僅有 Runge-Kutta 積分法能維持模擬的穩定性, 其餘三種方法皆出現爆震或數值發散的情況, 導致果凍模擬異常。然而, 當 delta time 進一步增大時, 即使是 Runge-Kutta 也無法避免不穩定現象的發生, 顯示即便是高階積分器, 在面對過大時間間隔時仍存在數值穩定性的限制。