

# CASE HW1

109550135 范恩宇

## 1. Introduction :

這次作業是要模擬軟體 ( 蒟蒻 ) 落下並撞擊不同類型表面後，因為不同的接觸方式、時間等等，從而產生不同的反彈方式、速度等物理現象。

## 2. Fundamentals :

作業中用到的知識說實在不難，大多都是國高中物理的內容，那以外的部分只要套上講義/作業教學上的公式到程式裡也能完成大綱。

首先的蒟蒻是用三種彈簧連接其中粒子，會用到 Spring force (  $F = -kx$  ) 和 Damper force (  $F = -cv$  )，兩者結合得出蒟蒻的 Internal force，從而模擬出蒟蒻撞到東西時應有的「Q 彈表現」。

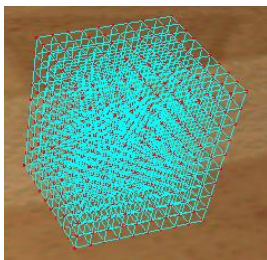
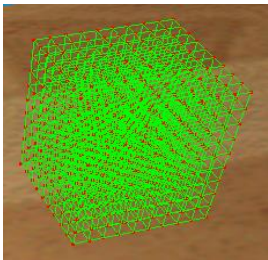
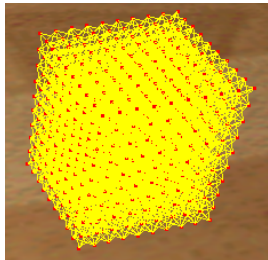
蒟蒻的粒子和平面/碗面碰撞，用的是一維碰撞公式，其中把蒟蒻粒子的速度分為 Normal velocity 和 Tangent velocity。先藉由碰撞面的法向量取得 Normal velocity，原速度值減去前者後得到 Tangent velocity，再藉公式  $V' = -k_r v_N + v_T$  得到新的速度。接著依據所受的外力 ( 重力 ) 和兩種力公式 (  $f^c = -(N \cdot f) N$  和  $f^f = -k_f (-N \cdot f) v_T$  ) 得到 contact force 與 friction force。

最後是用來讓動畫依據時間產生變化的積分器。Explicit 只需要最基本的  $x' = x + vt$  和  $v' = v + at$ 。Implicit、Midpoint 和 Runge Kutta 4th 則要先記下原先粒子的資訊，中間再依據個別要求將改變過的資訊乘上變化時間長，並與原先粒子資訊結合來得到新的，像 Implicit 用的中間資訊是「下次的」，Midpoint 用的是「這次與下次中間的」，Runge Kutta 4th 則將是下次的資訊「分成四份並給每份乘上特定權重」。

### 3. Implementation :

- Jelly

首先在 `initializeSpring()` 中實作 3 種不同的彈簧來將粒子們連起來，基於它們個別的特性，Stretch/Struct 和 Bend 實作 X、Y、Z 三個方向，Shear 則依據 XYZ 三軸的有無與正負實作  $1(\text{三軸全正}) + 3(\text{三軸有一負}) + 2 \times 3(\text{二軸二正/一負}) = 10$  個方向。

| Stretch/Struct  | Bend  | Shear   |
|---|---|---|
|  |  |  |

接著在 `computeSpringForce()` 與 `computeDamperForce()` 中分如「2.Fundamentals」中照著公式刻得到 spring force 與 damper

force 的方法。最後在 computeInternalForce() 中 traverse 彈簧們並得到個別的初始(末端)位置和速度，從而由前面的兩種 function 得到正反向的 spring force 和 damper force 並加到彈簧上。

```
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();
    Eigen::Vector3f delta_x = positionA - positionB;

    spring_force = -springCoef * (delta_x.norm() - restLength) * delta_x / delta_x.norm();

    return spring_force;
}

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 delta_x = positionA - positionB;
    Eigen::Vector3f delta_v = velocityA - velocityB;

    damper_force = -damperCoef * ((delta_v.dot(delta_x) / delta_x.norm()) * (delta_x / delta_x.norm()));

    return damper_force;
}

void Jelly::computeInternalForce() {
    // TODO#2-3: Compute the internal force (including spring force and damper force) for each spring.
    // 1. Read the start-particle and end-particle index from spring.
    // 2. Use 'getPosition()' to get particle i's position and use 'getVelocity()' get particle i's velocity.
    // 3. Call 'computeSpringForce' and 'computeDamperForce' to compute spring force and damper force.
    // 4. Compute net internal force and call 'addForce' to apply the force onto particles.
    // Note:
    // 1. Direction of the force.
    for (int i = 0; i < getSpringNum(); i++) {
        Spring sp = getSpring(i);

        Eigen::Vector3f start_p = particles[sp.getSpringStartID()].getPosition();
        Eigen::Vector3f start_v = particles[sp.getSpringStartID()].getVelocity();
        Eigen::Vector3f end_p = particles[sp.getSpringEndID()].getPosition();
        Eigen::Vector3f end_v = particles[sp.getSpringEndID()].getVelocity();

        Eigen::Vector3f spring_f = computeSpringForce(start_p, end_p, sp.getSpringCoef(), sp.getSpringRestLength());
        Eigen::Vector3f damper_f = computeDamperForce(start_p, end_p, start_v, end_v, sp.getDamperCoef());

        particles[sp.getSpringStartID()].addForce(spring_f);
        particles[sp.getSpringStartID()].addForce(damper_f);
        particles[sp.getSpringEndID()].addForce(-1*spring_f);
        particles[sp.getSpringEndID()].addForce(-1*damper_f);
    }
}
```

## ● Terrain

依據碰撞面種類，做不同的碰撞操作

### Plain：

基本上就照上課講義內容，先對 normal 與 particle 到平面的向量做內積，若小於 epsilon 則表示接近平面。如果這時速度 velocity 與 normal 的內積也小於 0，就會碰撞。此外，為避免 plain 處的作力和 bowl 處重疊，要讓 particle 到 hole\_position 的距離大於 bowl 的半徑才得以施加 plain 處的力。

對於符合前述全部條件的 particle，就照著講義上的公式算出  $v_n$  和  $v_t$ ，進而得到碰撞後的速度、contact force 與 friction force。如果 normal 和 particle 受到的力內積後小於 0，則將其施加到 particle。

## Bowl :

由於要確保蒟蒻在碗的範圍內才受到碗對應的作用力，先求出 particle 到 center 的距離，若小於 bowl class 中的 radius 且 particle 的 Y 座標低於 plain 平面，則進行以下步驟。

先判斷 radius 與 particle 到碗面向量的差值是否小於 epsilon，以及速度 velocity 和 normal 的內積是否小於 0，結果均為是則表示接近碗面並碰撞。

對於符合前述全部條件的 particle，一樣照講義上的公式分別算出  $v_n$  和  $v_t$ ，進而得到碰撞後的速度（用作業投影片上 bowl 對應的新公式）、contact force、friction force、碗面施予 particle 的速度，並將其施加到 particle 上。

```
void PlaneTerrain::handleCollision(const float delta_t, Jelly& jelly) {
    constexpr float EPSILON = 0.01f;
    constexpr float coeffResist = 0.8f;
    constexpr float coeffFriction = 0.2f;

    Eigen::Vector3f N_force = this->normal / this->normal.norm();
    for (int i = 0; i < jelly.getParticleNum(); i++) {
        // close to the wall's midpoint in
        Particle& particle = jelly.getParticle(i);
        Eigen::Vector3f pos = particle.getPosition();
        Eigen::Vector3f velocity = particle.getVelocity();
        // Eigen::Vector3f force = particle.getForce();
        if (abs(N_force.dot(pos - this->position)) < EPSILON && N_force.dot(velocity) < 0 && (pos - this->hole_position).norm() > this->hole_radius) {
            Eigen::Vector3f vn = velocity.dot(N_force) * N_force;
            Eigen::Vector3f vt = velocity - vn;
            Eigen::Vector3f v_new = -coeffResist * vn + vt;
            particle.setVelocity(v_new);

            if (N_force.dot(particle.getForce()) < 0) {
                Eigen::Vector3f contact_f = -N_force.dot(particle.getForce()) * N_force;
                particle.addForce(contact_f);
                Eigen::Vector3f friction_f = -coeffFriction * (-N_force.dot(particle.getForce()) * v0);
                particle.addForce(friction_f);
            }
        }
    }
}
```

```

void BowlTerrain::handleCollision(const float delta_T, Jelly& jelly) {
    constexpr float EPSILON = 0.01f;
    constexpr float coeffResist = 0.8f;
    constexpr float coeffFriction = 0.2f;

    for (int i = 0; i < jelly.getParticleNum(); i++) {
        Particle& particle = jelly.getParticle(i);
        Eigen::Vector3f velocity = particle.getVelocity();
        Eigen::Vector3f g_force = Eigen::Vector3f(0, -9.8f, 0);
        Eigen::Vector3f normal = (particle.getPosition() - position).normalized();
        Eigen::Vector3f N_force = normal / normal.norm();

        float hole_radius = this->radius / sqrt(2);
        float hole_dist = pow(particle.getPosition()(0) - HOLE_X, 2) + pow(particle.getPosition()(2) - HOLE_Z, 2); // since position = (HOLE_X, radius / sqrt(2), HOLE_Z)
        if (hole_dist <= pow(hole_radius, 2) && particle.getPosition()(1) <= 0) { // if in range of "half sphere", bowl && position.y = plain.y

            if (this->radius - (particle.getPosition() - position).norm() < EPSILON) { // distance < epsilon => collision
                // Eigen::Vector3f N_force = (pos - this->position) / (pos - this->position).norm();
                if (N_force.dot(velocity) > 0) {
                    Eigen::Vector3f vn = velocity.dot(N_force) * N_force;
                    Eigen::Vector3f vt = velocity - vn;
                    // bowl collision
                    Eigen::Vector3f v_new = -coeffResist * (vn * (particle.getMass() - this->mass) / (particle.getMass() + this->mass)) + vt;
                    particle.setVelocity(v_new);
                    Eigen::Vector3f contact_f = -N_force.dot(particle.getForce()) * N_force;
                    particle.addForce(contact_f);
                    Eigen::Vector3f friction_f = -coeffFriction * (-N_force.dot(particle.getForce()) * vt); //
                    particle.addForce(friction_f);

                    float N_bowl = (particle.getMass() + g_force).dot(normal);
                    particle.addVelocity(N_bowl * normal);
                }
            }
        }
    }
}

```

## ● Integrator

### Explicit Euler :

利用公式  $x' = x + vt$  與  $v' = v + at$ ，以現有的速度與加速度來分別

更新位置與速度。

```

void ExplicitEulerIntegrator::integrate(MassSpringSystem& particleSystem) {
    int particle_ct = particleSystem.getJellyPointer(0)->getParticleNum(); // jelly index = 0
    for (int i = 0; i < particle_ct; i++) {
        Particle* single_p = &particleSystem.getJellyPointer(0)->getParticle(i);

        // a' = g + f/m
        Eigen::Vector3f accel = particleSystem.gravity + (single_p->getForce() / single_p->getMass());
        Eigen::Vector3f pos = single_p->getPosition();
        Eigen::Vector3f velocity = single_p->getVelocity();
        float delta_t = particleSystem.deltaTime;

        // Integrate position first, x' = x + vt
        pos += velocity * delta_t;
        // Integrate velocity, v' = v + at
        velocity += accel * delta_t;

        // set new position & velocity
        single_p->setAcceleration(accel);
        single_p->setPosition(pos);
        single_p->setVelocity(velocity);
        // since acceleration is added
        single_p->setForce(Eigen::Vector3f::Zero());
    }
}

```

### Implicit Euler :

先用一個 vector 存下原本 particles 的資訊，再結合「下一個

particle」的資訊，套用和 explicit euler 中相同的位置與速度公式，得

到新位置與速度。

```

void ImplicitEulerIntegrator::integrate(MassSpringSystem& particleSystem) { // particleSystem here

int particle_ct = particleSystem.getJellyPointer(0)->getParticleNum(); // jelly index = 0
std::vector<Particle> previous_p;

for (int i = 0; i < particle_ct; i++) {
    previous_p.push_back(particleSystem.getJellyPointer(0)->getParticle(i));
}

for (int i = 0; i < particle_ct; i++) {
    Particle* single_p = &particleSystem.getJellyPointer(0)->getParticle(i);

    // a' = g + f/m
    Eigen::Vector3f accel = particleSystem.gravity + (single_p->getForce() / single_p->getMass());
    Eigen::Vector3f pos = single_p->getPosition();
    Eigen::Vector3f velocity = single_p->getVelocity();
    float delta_t = particleSystem.deltaTime;

    // Integrate position first, x' = x + vt
    pos += velocity * delta_t;
    // Integrate velocity, v' = v + at
    velocity += accel * delta_t;
    // set new position & velocity
    single_p->setAcceleration(accel);
    single_p->setPosition(pos);
    single_p->setVelocity(velocity);
    // since acceleration is added
    single_p->setForce(Eigen::Vector3f::Zero());
}

for (int it = 0; it < 2; it++) { // make original data used, as the formula
    particleSystem.computeJellyForce(*particleSystem.getJellyPointer(0));
    for (int i = 0; i < particle_ct; i++) {
        Particle* single_p = &particleSystem.getJellyPointer(0)->getParticle(i);

        float delta_t = particleSystem.deltaTime;
        // a' = g + f/m
        Eigen::Vector3f accel = particleSystem.gravity + (single_p->getForce() / single_p->getMass());
        Eigen::Vector3f pos = previous_p[i].getPosition() + single_p->getVelocity() * delta_t;
        Eigen::Vector3f velocity = previous_p[i].getVelocity() + accel * delta_t;

        single_p->setAcceleration(accel);
        // set new position & velocity
        single_p->setPosition(pos);
        single_p->setVelocity(velocity);
        // since acceleration is added
        single_p->setForce(Eigen::Vector3f::Zero());
    }
}
}

```

## Midpoint Euler :

同樣用一個 vector 存下原本 particles 的資訊，但這次結合「下一個 particle 與當前 particle 正中間」的資訊，用和前面兩種積分器相同的位置與速度公式，得到新位置與速度。

```

void MidpointEulerIntegrator::integrate(MassSpringSystem& particleSystem) {

int particle_ct = particleSystem.getJellyPointer(0)->getParticleNum(); // jelly index = 0
std::vector<Particle> previous_p;

for (int i = 0; i < particle_ct; i++) {
    previous_p.push_back(particleSystem.getJellyPointer(0)->getParticle(i));
}

for (int i = 0; i < particle_ct; i++) {
    Particle* single_p = &particleSystem.getJellyPointer(0)->getParticle(i);

    // a' = g + f/m
    Eigen::Vector3f accel = particleSystem.gravity + (single_p->getForce() / single_p->getMass());
    Eigen::Vector3f pos = single_p->getPosition();
    Eigen::Vector3f velocity = single_p->getVelocity();
    float delta_t = particleSystem.deltaTime;

    // Integrate position first, x' = x + vt
    pos += (velocity * delta_t)/2;
    // Integrate velocity, v' = v + at
    velocity += (accel * delta_t)/2;
    // set new position & velocity
    single_p->setAcceleration(accel);
    single_p->setPosition(pos);
    single_p->setVelocity(velocity);
    // since acceleration is added
    single_p->setForce(Eigen::Vector3f::Zero());
}

particleSystem.computeJellyForce(*particleSystem.getJellyPointer(0));
for (int i = 0; i < particle_ct; i++) {
    Particle* single_p = &particleSystem.getJellyPointer(0)->getParticle(i);

    float delta_t = particleSystem.deltaTime;
    // a' = g + f/m
    Eigen::Vector3f accel = particleSystem.gravity + (single_p->getForce() / single_p->getMass());
    Eigen::Vector3f pos = previous_p[i].getPosition() + single_p->getVelocity() * delta_t;
    Eigen::Vector3f velocity = previous_p[i].getVelocity() + accel * delta_t;

    single_p->setAcceleration(accel);
    // set new position & velocity
    single_p->setPosition(pos);
    single_p->setVelocity(velocity);
    // since acceleration is added
    single_p->setForce(Eigen::Vector3f::Zero());
}
}

```

## Runge Kutta 4th:

同樣需要用一個 vector 存下原本 particles 的資訊，然而這次要依據公式要求，將原本的計算過程分為 k1,2,3,4 共 4 階段且各自乘上對應的權重，並把各階段的位置與速度變量整合至 particle。

```
void RungeKuttaFourthIntegrator::integrate(MassSpringSystem& particleSystem) {  
    struct StateStep {  
        Eigen::Vector3f deltaVel;  
        Eigen::Vector3f deltaPos;  
    };  
  
    StateStep step;  
    int particle_ct = particleSystem.getJellyPointer(0)->getParticleNum(); // jelly index = 0  
    std::vector<Particle> previous_p;  
    std::vector<Particle> new_p;  
  
    for (int i = 0; i < particle_ct; i++) {  
        previous_p.push_back(particleSystem.getJellyPointer(0)->getParticle(i));  
        new_p.push_back(particleSystem.getJellyPointer(0)->getParticle(i));  
    }  
  
    // k1  
    for (int i = 0; i < particle_ct; i++) {  
        Particle* single_p = &particleSystem.getJellyPointer(0)->getParticle(i);  
  
        // a' = g + f/m  
        Eigen::Vector3f accel = particleSystem.gravity + (single_p->getForce() / single_p->getMass());  
        Eigen::Vector3f pos = single_p->getPosition();  
        Eigen::Vector3f velocity = single_p->getVelocity();  
        float delta_t = particleSystem.deltaTime;  
  
        // Integrate position first, x' = x + vt  
        step.deltaPos = velocity * delta_t;  
        // Integrate velocity, v' = v + at  
        step.deltaVel = accel * delta_t;  
  
        // set new position & velocity  
        single_p->setAcceleration(accel);  
        //single_p->setPosition(pos);  
        single_p->setVelocity(previous_p[i].getVelocity() + step.deltaVel);  
        // since acceleration is added  
        single_p->setForce(Eigen::Vector3f::Zero());  
        new_p[i].addPosition(step.deltaPos / 6.0);  
        new_p[i].addVelocity(step.deltaVel / 6.0);  
    }  
  
    // k2  
    particleSystem.computeJellyForce(*particleSystem.getJellyPointer(0));  
    for (int i = 0; i < particle_ct; i++) {  
        Particle* single_p = &particleSystem.getJellyPointer(0)->getParticle(i);  
  
        // a' = g + f/m  
        Eigen::Vector3f accel = particleSystem.gravity + (single_p->getForce() / single_p->getMass());  
        Eigen::Vector3f pos = single_p->getPosition();  
        Eigen::Vector3f velocity = single_p->getVelocity();  
        float delta_t = particleSystem.deltaTime;  
  
        // Integrate position first, x' = x + vt  
        step.deltaPos = velocity * delta_t;  
        // Integrate velocity, v' = v + at  
        step.deltaVel = accel * delta_t;  
  
        // set new position & velocity  
        single_p->setAcceleration(accel);  
        //single_p->setPosition(pos);  
        single_p->setVelocity(previous_p[i].getVelocity() + 0.5 * step.deltaVel);  
        // since acceleration is added  
        single_p->setForce(Eigen::Vector3f::Zero());  
        new_p[i].addPosition(step.deltaPos / 3.0);  
        new_p[i].addVelocity(step.deltaVel / 3.0);  
    }  
  
    // k3  
    particleSystem.computeJellyForce(*particleSystem.getJellyPointer(0));  
    for (int i = 0; i < particle_ct; i++) {  
        Particle* single_p = &particleSystem.getJellyPointer(0)->getParticle(i);  
  
        // a' = g + f/m  
        Eigen::Vector3f accel = particleSystem.gravity + (single_p->getForce() / single_p->getMass());  
        Eigen::Vector3f pos = single_p->getPosition();  
        Eigen::Vector3f velocity = single_p->getVelocity();  
        float delta_t = particleSystem.deltaTime;  
  
        // Integrate position first, x' = x + vt  
        step.deltaPos = velocity * delta_t;  
        // Integrate velocity, v' = v + at  
        step.deltaVel = accel * delta_t;  
  
        // set new position & velocity  
        single_p->setAcceleration(accel);  
        //single_p->setPosition(pos);  
        single_p->setVelocity(previous_p[i].getVelocity() + 0.5 * step.deltaVel);  
        // since acceleration is added  
        single_p->setForce(Eigen::Vector3f::Zero());  
        new_p[i].addPosition(step.deltaPos / 3.0);  
        new_p[i].addVelocity(step.deltaVel / 3.0);  
    }  
}
```

```

// k4
particleSystem.computeJellyForce(*particleSystem.getJellyPointer(0));
for (int i = 0; i < particle_ct; i++) {
    Particle* single_p = &particleSystem.getJellyPointer(0)->getParticle(i);

    // a' = g + f/m
    Eigen::Vector3f accel = particleSystem.gravity + (single_p->getForce() / single_p->getMass());
    Eigen::Vector3f pos = single_p->getPosition();
    Eigen::Vector3f velocity = single_p->getVelocity();
    float delta_t = particleSystem.deltaTime;

    // Integrate position first, x' = x + vt
    step.deltaPos = velocity * delta_t;
    // Integrate velocity, v' = v + at
    step.deltaVel = accel * delta_t;

    // set new position & velocity
    single_p->setAcceleration(accel);
    //single_p->setPosition(pos);
    single_p->setVelocity(previous_p[i].getVelocity() + step.deltaVel);//
    // since acceleration is added
    single_p->setForce(Eigen::Vector3f::Zero());
    new_p[i].addPosition(step.deltaPos / 6.0);
    new_p[i].addVelocity(step.deltaVel / 6.0);
}

// update
for (int i = 0; i < particle_ct; i++) {
    Particle* single_p = &particleSystem.getJellyPointer(0)->getParticle(i);
    Eigen::Vector3f pos = new_p[i].getPosition();
    Eigen::Vector3f velocity = new_p[i].getVelocity();
    single_p->setPosition(pos);
    single_p->setVelocity(velocity);
    single_p->setForce(Eigen::Vector3f::Zero());
}

```

## 4. Result and Discussion :

### ● The difference between integrators

四者間最明顯的差異大概在於運動速度，尤其一開始下落及第一次的「部分蒟蒻衝出碗邊界並滑回碗內」時，其他時刻的差異不明顯。

Explicit 會進行地最快，應該是正常速度？Implicit 相較前者慢了一點，大概是因為中間有先算下次資訊來跟當前的結合，並算出更新值。

而 Midpoint 更慢，因為要讓它跑到一半，再如同 Implicit 先算下次資訊再跟跑到一半的結合算出更新值，需要耗費更多的計算量。Runge Kutta 4th 要做四次類似 Implicit 與 Midpoint 的計算來得出四塊，顯然會比 Midpoint 的兩塊還要更慢且幀數更低。



## ● Effect of parameters

`springCoef` 是彈簧的彈力係數，其數值越大則彈簧的張力越大，形狀會越難以改變；相對地，彈力係數越小則彈簧本身越鬆散，蒟蒻受到形變後會更難以恢復原狀。

`damperCoef` 是阻尼係數，用來讓彈簧停止。彈簧振動時，會受到阻尼力的影響，逐漸減少振幅直到停止。該係數數值越大則彈簧越穩定且容易停止振動；相對地，阻尼係數越小則彈簧彈性越強，即使受到的力少，也能較長時間地持續震動。

`coefResist` 是接觸力的係數。物體碰撞時，該數值越大則物體受到的反作用力越大；反之，受到的反作用力越小，使回彈的幅度降低。

`coefFriction` 是摩擦力的係數。物體在面上滑行時，該數值越大則同物體受到的摩擦力越大，會越早停止滑行；反之，受到的摩擦力越小，物體越容易維持滑行狀態。

`deltaTime` 用來表示時間的變化量。值越大則可以使運行更加平滑，因為計算量較少，但較大的 `deltaTime` 也會使動畫略微變得粗糙。不過，只要該值不會太大，依舊能利用視覺暫留的效果產生動畫效果；反之，太大的 `deltaTime` 容易使部分功能無法正常運作，當然也產生不出正確的結果。

## 5. Conclusion :

雖然助教們已經搞定了整個 project 大部分的東西，這次作業 sample code 的架構說實在有點複雜，所以一開始花了不少時間確定各個參數的來源以及用處。尤其某些 TODO 長一模一樣但實際該進行不同操作處 ( ex : 3-1&3-2、4-2&4-3 ) 真的做很卡，真的不是套個公式就能輕鬆做出來，深刻瞭解到為什麼教授建議會我們早點開始做 XD

不過幸虧有討論區同學與助教們的踴躍發問與熱心解答，遇上的問題幾乎都能得到有效的解答，讓我能順利完成作業。然後如果可以的話，覺得能把討論區一些「討論度較高」的討論串結論另行公告，感覺不少人都沒在看就直接開問，或許可以減少類似的問題洗版？