2020 CA HW1 Report

一、 作業目標

本次作業目的在於做出真實的物理模擬，藉由簡單的下落與環境的碰撞，來初步了解動畫中的模擬數學式。
主要架構已經準備好，我們只需要專注於實作需要的數學式即可。

二、 環境

IDE: Visual studio 2017
Platform: Windows
Graphics API: OpenGL
OpenGL Loading Library: glad
OpenGL Toolkit: glfw
UI Library: dear imgui
Math Library: Eigen

- ➔ Eigen::Vector3f

三、 作業內容

1. 建立物體中的彈簧系統
2. 計算彈簧的彈簧力與阻尼力
3. 處理四種地形的碰撞結果
    a. 平面
    b. 球體外部
    c. 球體內部（碗型）
    d. 斜面

4. 實作四種不同的積分器
    a. Explicit Euler
    b. Implicit Euler
    c. Midpoint Euler
    d. Runge Kutta 4^{th}

四、 實作

1. 建立彈簧系統

由於方塊中的所有粒子都已經建立好，因此我們只需要建立其中相連結的彈簧部分即可。

彈簧分為三種：

Structure spring：與三軸上最近的 6 個點相連
Bending spring：與三軸上最近的 6 個點隔一格的六個點相連
（目的在於防止 Structure springs 在受力後不正常扭曲扳折）
Shear spring：與三維空間中，對角線的 20 個點相連

每個彈簧都會連接兩個點，因此實際上每個點都只需要負責處理一半的彈簧量。
Structure spring 與 Bending spring 都只需要負責軸上正向的 3 個點即可。
最困難的是 Shear spring，必須找到 10 個方向全都不同的彈簧。

以下實作以 Structure spring 為例：

```
std::vector<int> Struct_x;
std::vector<int> Struct_y;
std::vector<int> Struct_z;

if (i < particleNumPerEdge - 1) {
    Struct_x.push_back(i + 1);
    Struct_y.push_back(j);
    Struct_z.push_back(k);
}

if (j < particleNumPerEdge - 1) {
    Struct_x.push_back(i);
    Struct_y.push_back(j + 1);
    Struct_z.push_back(k);
}
```

1. 將所有需要的鄰居點的 particle number 推入 vector 中。
2. 若為最外側的平面或邊或角落可能就不到三條彈簧。
3. 將自己的 particle number 與 vector 所記錄的 neighboring particle number 作為彈簧的兩個頂點，並生出一個彈簧，推入儲存著所有彈簧的 vector 中。

```
for (int t = 0; t < Struct_x.size(); t++) {
    iNeighborID = Struct_x[t] * particleNumPerFace + Struct_y[t] * particleNumPerEdge + Struct_z[t];

    SpringStartPos = particles[iParticleID].getPosition();
    SpringEndPos = particles[iNeighborID].getPosition();
    Length = (SpringStartPos - SpringEndPos).norm();

    Spring Spring(iParticleID, iNeighborID, Length, springCoef, damperCoef, Spring::SpringType::STRUCT);
    springs.push_back(Spring);
}
```

2. **計算彈簧力與阻尼力**

計算所有彈簧的彈簧力與阻尼力。

彈簧力取決於彈簧長度與原始長度的差異，與初始長度差愈多，彈簧力則愈大。

$$\vec{f}_a = -k_s(|\vec{x}_a - \vec{x}_b| - r)\frac{\vec{x}_a - \vec{x}_b}{|\vec{x}_a - \vec{x}_b|}$$

阻尼力則取決於長度與速度。阻尼力是為了阻止彈簧無限來回彈，讓他最終會回歸初始。

$$\vec{f}_a = -k_d \frac{(\vec{v}_a - \vec{v}_b)\cdot(\vec{x}_a - \vec{x}_b)}{|\vec{x}_a - \vec{x}_b|}\frac{(\vec{x}_a - \vec{x}_b)}{|\vec{x}_a - \vec{x}_b|}$$

最後只需要將兩者都加上彈簧兩端的粒子上。

※要注意的是雖然算出來的結果是彈簧力與阻尼力，看似都是作用在彈簧上，但實際上要將此力施加在粒子上，並且兩端粒子受到的力是大小相等，方向相反。

計算彈簧本身的所有力，並加結果加在兩端的粒子上：

```
Eigen::Vector3f SpringForce = computeSpringForce(StartParticle.getPosition(), EndParticle.getPosition(),
                                                 Spring.getSpringCoef(), Spring.getSpringRestLength());

Eigen::Vector3f DamperForce = computeDamperForce(StartParticle.getPosition(), EndParticle.getPosition(),
                                                 StartParticle.getVelocity(), EndParticle.getVelocity(),
                                                 Spring.getDamperCoef());

StartParticle.addForce(SpringForce);
StartParticle.addForce(DamperForce);
particles[Spring.getSpringStartID()] = StartParticle;

EndParticle.addForce(-SpringForce);
EndParticle.addForce(-DamperForce);
particles[Spring.getSpringEndID()] = EndParticle;
```

計算彈簧力：

```
Eigen::Vector3f relativePosition = positionA - positionB;
float currentLength = relativePosition.norm();

Eigen::Vector3f normalizedRelativePosition = relativePosition / currentLength;

Eigen::Vector3f SpringForce = -springCoef * ((currentLength - restLength)) * normalizedRelativePosition;

return SpringForce;
```

計算阻尼力：

```
Eigen::Vector3f relativePosition = positionA - positionB;
Eigen::Vector3f relativeVelocity = velocityA - velocityB;
float currentLength = relativePosition.norm();

Eigen::Vector3f normalizedRelativePosition = relativePosition / currentLength;

Eigen::Vector3f DamperForce = -damperCoef * (relativeVelocity.dot(normalizedRelativePosition)) * normalizedRelativePosition;

return DamperForce;
```

3. **處理四種地形的碰撞結果**

四種處理方式非常相似，尤其是平面與斜面，基本上一模一樣，而球型與碗型則也非常相似，只是判斷式相反。

需要做的判斷如下（以下碰撞面都以牆面代稱）：

1. 判斷是否有發生碰觸（距離 $< \varepsilon$ ）

   平面／斜面：

   ```
   if ((currentParticle.getPosition() - position).dot(normal) < eEPSILON) {
   ```

   球型：

   ```
   if ((centerToParticleLength - radius) < eEPSILON) {
   ```

   碗型：

   ```
   if ((radius - centerToParticleLength) < eEPSILON) {
   ```

2. 速度朝向牆

   ```
   if (currentParticle.getVelocity().dot(normal) < 0) {
   ```

3. 粒子當下受到的力的總和朝向牆面

   ```
   if (currentForce.dot(normal) < 0) {
   ```

需要做的應對如下：

1. 若有發生碰觸，才進入以下環節
2. 若速度朝向牆面，則計算碰撞後的速度，並直接更新該粒子的速度值
3. 若粒子當下受到的合力朝向牆面，則將該法向量方向的力完全抵消，並額外計算摩擦力。

對於四種不同的平面所需要做的處理，差別有二：

1. 對於平面與斜面來說，法向量在牆面各處都一樣，而球型、碗型的法向量則取決於粒子碰到的位置，因此需要額外計算粒子向著球心方向來得到法向量。

   ```
   Eigen::Vector3f terrainNormalVector = currentParticle.getPosition() - position;
   float centerToParticleLength = terrainNormalVector.norm();
   terrainNormalVector = terrainNormalVector / centerToParticleLength;
   ```

2. 球型與碗型的碰撞後速度，其碰撞參數取決於粒子與球體的質量，平面與斜面因無質量參數，因此需要另外給定碰撞參數。

   平面／斜面：

```
if (currentParticle.getVelocity().dot(normal) < 0) {
    Eigen::Vector3f newVelocity = -resistCoef * normalVelocity + tangentialVelocity;
    cube.getParticle(i).setVelocity(newVelocity);
}
```

球型／碗型：

```
Eigen::Vector3f newVelocity =
    ((particleMass - mass) / (particleMass + mass)) * normalVelocity + tangentialVelocity;
cube.getParticle(i).setVelocity(newVelocity);
```

以下以平面為例：

```
for (int i = 0; i < cube.getParticleNum(); i++) {

    Particle currentParticle = cube.getParticle(i);

    if ((currentParticle.getPosition() - position).dot(normal) < eEPSILON) {

        float normalLength = normal.norm();

        Eigen::Vector3f totalVelocity = currentParticle.getVelocity();
        Eigen::Vector3f normalVelocity = (totalVelocity.dot(normal) / (normalLength * normalLength)) * normal;
        Eigen::Vector3f tangentialVelocity = totalVelocity - normalVelocity;

        if (currentParticle.getVelocity().dot(normal) < 0) {
            Eigen::Vector3f newVelocity = -resistCoef * normalVelocity + tangentialVelocity;
            cube.getParticle(i).setVelocity(newVelocity);
        }

        Eigen::Vector3f currentForce = currentParticle.getForce();

        if (currentForce.dot(normal) < 0) {
            Eigen::Vector3f collisionForce = -currentForce.dot(normal) * normal;
            Eigen::Vector3f frictionalForce =
                frictionCoef * currentForce.dot(normal) * (tangentialVelocity / tangentialVelocity.norm());

            cube.getParticle(i).addForce(collisionForce);
            cube.getParticle(i).addForce(frictionalForce);
        }
    }
}
```

4. **實作四種不同的積分器**

四種積分器分別是 Explicit Euler、Implicit Euler、Midpoint Euler 以及 Runge Kutta 4th。

Explicit Euler：

非常好理解，以位置為例，當下的速度乘上時間區間，即為位移量。

$$\mathbf{x}_{n+1} = \mathbf{x}_n + hf(\mathbf{x}_n, t_n)$$

此處的 x 可以為位置或速度，而 h 則對應到 Δt。f(x, t)則為變量 x 對時間 t 的微分。

此方法雖然非常簡單易懂，但極為不精確，原因是在電腦模擬上，不可能將時間區間 Δt 取至無限小，因此每過一個時間區間，誤差值將會被愈放愈大。

```
for (int i = 0; i < particleSystem.getCubeCount(); i++) {
    for (int j = 0; j < particleSystem.getCubePointer(i)->getParticleNum(); j++) {
        Particle currentParticle = particleSystem.getCubePointer(i)->getParticle(j);

        Eigen::Vector3f deltaPosition = currentParticle.getVelocity() * particleSystem.deltaTime;
        Eigen::Vector3f deltaVelocity = currentParticle.getAcceleration() * particleSystem.deltaTime;

        particleSystem.getCubePointer(i)->getParticle(j).addPosition(deltaPosition);
        particleSystem.getCubePointer(i)->getParticle(j).addVelocity(deltaVelocity);
    }
}
```

Implicit Euler：

若是將未來發生的變化，帶入當下的情況中，就可以縮小前期誤差造成的放大問題，這也正是 Implicit Euler 的做法。

$$\mathbf{x}_{n+1} = \mathbf{x}_n + hf(\mathbf{x}_{n+1}, t_{n+1})$$

此舉的缺點在於，我們必須得出下一個瞬間的變化量。從本次作業來看，意思就是我們必須計算出下一個瞬間該粒子的速度。

因此在算完第一次時間區間，到了未來之後，

```
for (int j = 0; j < currentCube->getParticleNum(); j++) {
    Particle currentParticle = currentCube->getParticle(j);

    startingPosition.push_back(currentParticle.getPosition());
    startingVelocity.push_back(currentParticle.getVelocity());

    deltaPosition = currentParticle.getVelocity() * particleSystem.deltaTime;
    deltaVelocity = currentParticle.getAcceleration() * particleSystem.deltaTime;

    currentParticle.setPosition(startingPosition[j] + deltaPosition);
    currentParticle.setVelocity(startingVelocity[j] + deltaVelocity);
}
```

計算下一個瞬間未來的新的速度與加速度，

```
particleSystem.computeCubeForce(*currentCube);
```

以此來得到這個瞬間的 Implicit Euler 變化量。

```
for (int j = 0; j < currentCube->getParticleNum(); j++) {
    Particle currentParticle = currentCube->getParticle(j);

    deltaPosition = currentParticle.getVelocity() * particleSystem.deltaTime;
    deltaVelocity = currentParticle.getAcceleration() * particleSystem.deltaTime;

    particleSystem.getCubePointer(i)->getParticle(j).setPosition(startingPosition[j] + deltaPosition);
    particleSystem.getCubePointer(i)->getParticle(j).setVelocity(startingVelocity[j] + deltaVelocity);
}
```

Midpoint Euler：

依照自身的理解，此方法與 Implicit Euler 非常相近，差別只在於，雖然我們計算出了 Explicit Euler 下的新的位置，但這次我們只取一半的時間，意即只取一半的變化量。到了這個「中間」的「未來」狀態時，再來計算新的變化量。並且最後由此新的變化量，作為現在實際上的變化狀況來進行更新。

從 Code 的角度上唯一的差別在於，Implicit Euler 到達的未來是 Δt 的未來，而 Midpoint Euler 只到達了 $\Delta t / 2$。

```
currentParticle.setPosition(startingPosition[j] + deltaPosition / 2);
currentParticle.setVelocity(startingVelocity[j] + deltaVelocity / 2);
```

Runge Kutta 4^{th}：

最後一個最難的，卻也是目前在物理模擬上，最常被使用的積分器，Runge Kutta 4^{th}。

此積分器的概念在於，既然只取當下狀況會造成很大誤差，只取一次未來似乎又可能造成另一個方向的誤差，那何不乾脆多取幾個未來，來取平均呢？只取平均甚至不夠，還是加權過的平均。

公式大致如下：

$$k_1 = hf(\mathbf{x}_0, t_0)$$
$$k_2 = hf(\mathbf{x}_0 + \frac{k_1}{2}, t_0 + \frac{h}{2})$$
$$k_3 = hf(\mathbf{x}_0 + \frac{k_2}{2}, t_0 + \frac{h}{2})$$
$$k_4 = hf(\mathbf{x}_0 + k_3, t_0 + h)$$

$$\mathbf{x}(t_0+h)=\mathbf{x}(t_0)+\frac{1}{6}(k_1+2k_2+2k_3+k_4)+O(h^5)$$

從前面的 Implicit Euler 與 Midpoint Euler 我們可以很快了解，這裡的 k1 到 k4，其實都是利用前一個 k 算出來的未來，所算得的「新的未來」，最後再用這四個不同的未來，加權平均，以此來逼近真實狀況。

從 code 上來說，就是如 Implicit Euler 與 Midpoint Euler，「計算未來」的部分執行四遍，將四次所算得的未來都記錄起來，最後取出來加權。

```
for (int j = 0; j < currentCube->getParticleNum(); j++) {
    Particle currentParticle = currentCube->getParticle(j);

    Eigen::Vector3f deltaPosition =
        (k1[j].deltaPos + 2 * k2[j].deltaPos + 2 * k3[j].deltaPos + k4[j].deltaPos) / 6;
    Eigen::Vector3f deltaVelocity =
        (k1[j].deltaVel + 2 * k2[j].deltaVel + 2 * k3[j].deltaVel + k4[j].deltaVel) / 6;
    particleSystem.getCubePointer(i)->getParticle(j).setPosition(startingPosition[j] + deltaPosition);
    particleSystem.getCubePointer(i)->getParticle(j).setVelocity(startingVelocity[j] + deltaVelocity);
}
```

這四種積分器由於本次作業的模擬狀況完全稱不上複雜，因此較難判斷何者較好，視覺上能夠看到的最大差異就是，Runge Kutta 4th 在計算上非常花時間，因此下落的速度較其他三個積分器都要來的慢。

五、 Bonus

為了解決方塊從過高的地方下落後，造成的方塊整體變形的狀況，因此在 initializeSpring 的部分，我新增了非常多 Bending Spring，全部都是從六面中的任何一個平面，直接延伸到對面去的超長彈簧。如此就更不容易產生方塊角落整個塌陷進方塊內部的問題。

```
if (i == 0) {
    Bend_x.push_back(particleNumPerEdge - 1);
    Bend_y.push_back(j);
    Bend_z.push_back(k);
}

if (j == 0) {
    Bend_x.push_back(i);
    Bend_y.push_back(particleNumPerEdge - 1);
    Bend_z.push_back(k);
}

if (k == 0) {
    Bend_x.push_back(i);
    Bend_y.push_back(j);
    Bend_z.push_back(particleNumPerEdge - 1);
}
```

六、 難點（疑問）與解法

本次實作有多個狀況產生。

1. 實作 handleTerrain

    (1.) 在實作此部分時，原本一直不了解為何碰撞時又要提供 v 又要提供 f，因此一直沒辦法消除起初方塊詭異跳動的狀況。直到後來終於理解速度才是真正使粒子更新

動態的主要功臣，所謂的 resist force 目的是要將往牆面中的力完全抵消，而提供的反方向力，以避免方塊繼續朝著牆壁移動。

(2.) 第二個大問題是摩擦力的實作，由於上課講義中摩擦力公式裡的 $\mathbf{v}_t$ 並沒有特別說明各項值的真實含意，因此原本只有照抄的我犯了極大的錯誤，就是該 $\mathbf{v}_t$必須是單位向量。至此，摩擦力才真正完整

2. 實作積分器失敗（Explicit Euler 除外）

一直認為是我對於剩餘三個積分器的理解發生錯誤，實際上是 handleTerrain 的部分也一直有問題沒有修正，因此在 Debug 時很難確定究竟誰才是問題主因。也為此上網查了更多有關積分器的資訊，也終於在最後完整了解了各項積分器之間的差異以及目的。

3. Bowl 的彈跳問題

此問題可以說是仍然沒有解決。

從助教的影片中可以看到，方塊在碗中完全不會彈跳，基本上一落地就只剩下滑行了。但在我的狀況中，方塊在碗裡還會先跳個一兩下，直到最後被摩擦力給定在中間，才會完全停下。至今仍不知道助教怎麼做到讓方塊完全沒有彈跳效果的。

從視覺上主觀來說，兩者看起來都滿不真實，因此最後我選擇維持原樣。

七、 結語

雖然對於物理與微分方程並不排斥，但真正到了實作時，實在也是耗腦耗時間。實在是幸好這次的功課只需要填入各個物理公式，而不需要自己處理 OpenGL 的顯示，讓我們可以專心在理解物理與微方上，而非花費時間在一堆瑣碎的 Code 上。只經過這麼一次作業就又覺得自己對於電腦動畫的執著又更上一層樓，期待下一次作業的我可以再次進步（並且不要花這麼多時間）。