Introduction

本次作業為實作粒子系統的物理引擎，使用彈簧將整齊排列的粒子串聯起來形成一個方塊物體，並且導入各式不同的力學公式模擬方塊使之達成如同現實中彈性物體的掉落碰撞反映，搭配 OpenGL 所繪製的 3D 模型可以直覺的看到模擬的效果。

Fundamentals

粒子系統：

在此專案中由 10*10*10 個粒子所組成，每個粒子有加速度(acceleration)、受力(force)、質量(mass)、位置(position)、速度(velocity)這些參數。

彈簧系統：

使用彈簧連結所有的粒子使其變成一個完成的立方體結構，每個彈簧有阻尼器參數(damperCoef)、彈簧參數(springCoef)、兩端連結的粒子(StartID&EndID)、彈簧原始長度(RestLength)、彈簧種類(struct/shear/bending) 。

每個彈簧會受到彈簧力與阻尼力分別如下:

彈簧力：

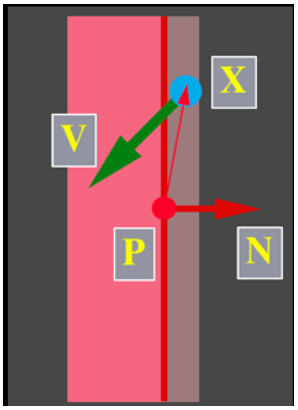
$$-k_s\left(\left|\vec{x}_a-\vec{x}_b\right|-r\right)\frac{\vec{x}_a-\vec{x}_b}{\left|\vec{x}_a-\vec{x}_b\right|}$$

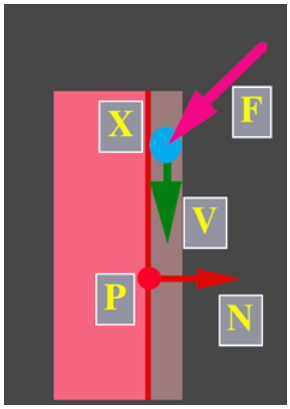
阻尼力：

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

碰撞：

在此專案中地形有四種，分別為 Plane, TiltedPlane, Sphere and Bowl，每一種都有不同的碰撞判定方式，碰撞的示意圖如講義所示：





積分器：

在此專案中積分器有四種，分別為Explicit Euler, Implicit Euler, Midpoint Euler, Runge-Kutta 4th，每一種方法用不同的數學算法去近似求解微分方程，進而得到粒子系統模擬的效果。大致上越複雜的方法的解就越精準，但需要更多的運算時間，設計上也更加複雜。如Runge-Kutta 4th，便是非常複雜但精準的積分器算法。

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

Implementation

Struct 彈簧連結：使用三層迴圈處理 Z-axis 之彈簧串聯，其他兩軸做法相同。

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

            //STRUCT spring
            iParticleID = i * particleNumPerFace + j * particleNumPerEdge + k;
            iNeighborID = i * particleNumPerFace + j * particleNumPerEdge + (k + 1);
            struct_spring[counter] = Spring(iParticleID, iNeighborID, struct_spring_length, springCoef, damperCoef,
                                    simulation::Spring::SpringType::STRUCT);
            springs.push_back(struct_spring[counter]);
            counter++;
        }
    }
}
```

Bending 彈簧連結：將 struct 連結部分 1 改為 2。

```
for (int i = 0; i < particleNumPerEdge; i++) {
    for (int j = 0; j < particleNumPerEdge; j++) {
        for (int k = 0; k < particleNumPerEdge - 2; k++) {
            // BENDING spring
            iParticleID = i * particleNumPerFace + j * particleNumPerEdge + k;
            iNeighborID = i * particleNumPerFace + j * particleNumPerEdge + (k + 2);
            bend_spring[counter] = Spring(iParticleID, iNeighborID, bend_spring_length, springCoef, damperCoef,
                                    simulation::Spring::SpringType::BENDING);
            springs.push_back(bend_spring[counter]);
            counter++;
        }
    }
}
```

Shear 彈簧連結：總共有 16 個方向，依次連結使用 16 個迴圈達成。

```
for (int i = 0; i < particleNumPerEdge; i++) {
    for (int j = 0; j < particleNumPerEdge-1; j++) {
        for (int k = 0; k < particleNumPerEdge-1; k++) {
            // SHEAR spring
            iParticleID = i * particleNumPerFace + j * particleNumPerEdge + k;
            iNeighborID = i * particleNumPerFace + (j+1) * particleNumPerEdge + (k + 1);
            shear_spring[counter] = Spring(iParticleID, iNeighborID, shear_spring_length, springCoef, damperCoef,
                                    simulation::Spring::SpringType::SHEAR);
            springs.push_back(shear_spring[counter]);
            counter++;
        }
    }
}
```

將建構完成的彈簧 push 至 springs 這個 vector 當中。

彈簧力：實作 fundamentals 提及之公式如下

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

阻尼力：實作 fundamentals 提及之公式如下

```
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 DF;
  Eigen::Vector3f positionBA = positionA - positionB;
  Eigen::Vector3f velocityBA = velocityA - velocityB;
  float distance =
     sqrt(positionBA[0] * positionBA[0] + positionBA[1] * positionBA[1] + positionBA[2] * positionBA[2]);
  float v_dot_x = velocityBA[0] * positionBA[0] + velocityBA[1] * positionBA[1] + velocityBA[2] * positionBA[2];
  DF = -1 * damperCoef * v_dot_x * positionBA / (distance*distance);
  return DF;
}
```

依序搜尋每一條彈簧並且計算每一條彈簧當下的受力。

```
void Cube::computeInternalForce() {
  // TODO

  Eigen::Vector3f SF;  // computeSpringForce
  Eigen::Vector3f DF;  // computeDamperForce

     for (int i = 0; i < springs.size(); i++) {


        SF = computeSpringForce(
           particles[springs[i].getSpringStartID()].getPosition(),
           particles[springs[i].getSpringEndID()].getPosition(),
           springs[i].getSpringCoef(),
           springs[i].getSpringRestLength()
        );

        DF = computeDamperForce(
           particles[springs[i].getSpringStartID()].getPosition(),
           particles[springs[i].getSpringEndID()].getPosition(),
           particles[springs[i].getSpringStartID()].getVelocity(),
           particles[springs[i].getSpringEndID()].getVelocity(),
           springs[i].getDamperCoef()
        );


        particles[springs[i].getSpringStartID()].addForce(SF+DF);          //彈簧力+阻尼力
        particles[springs[i].getSpringEndID()].addForce(-1* (SF+DF));  //彈簧力+阻尼力的反作用力

     }
}
```

平面碰撞

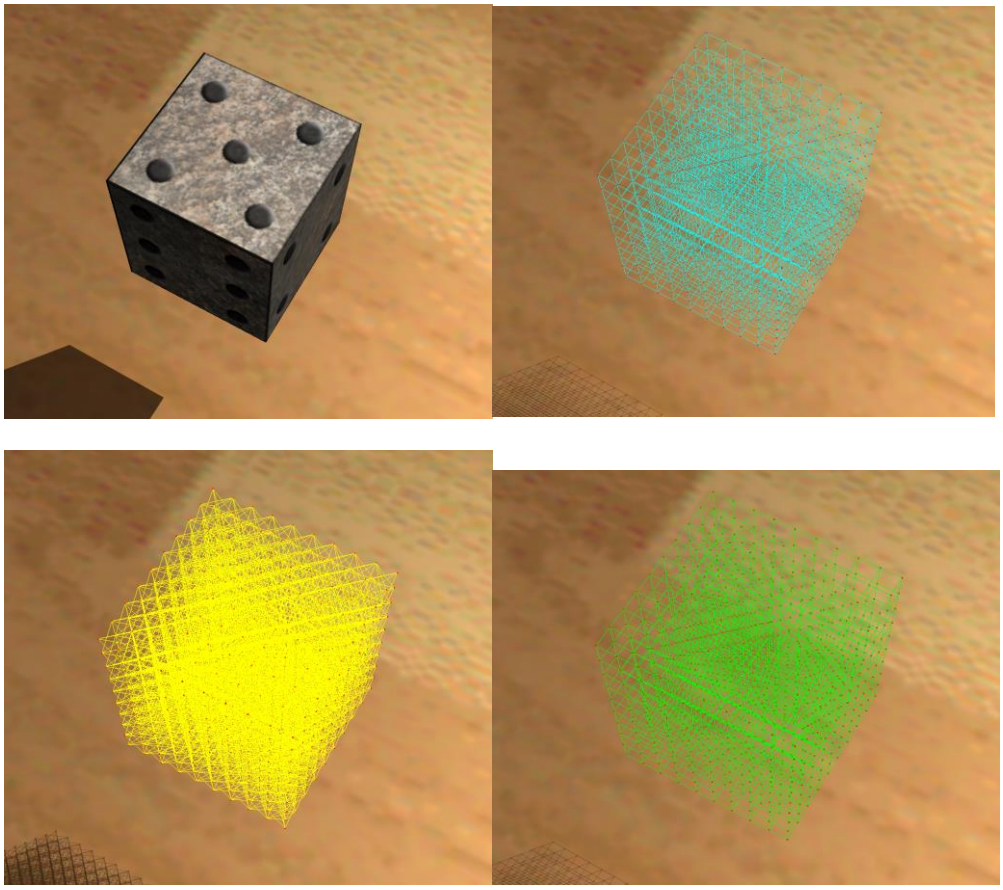
```cpp
void PlaneTerrain::handleCollision(const float delta_T, Cube& cube) {
    constexpr float eEPSILON = 0.01f;
    constexpr float coefResist = 0.8f; //kr
    constexpr float coefFriction = 0.3f; //kf
    // TODO
    int par_num = cube.getParticleNum();

    Eigen::Vector3f coll;
    Eigen::Vector3f fc;
    coll[0] = 0;
    coll[1] = 4.9f;
    coll[2] = 0;
    for (int i = 0; i < 1000; i += 1) {
        if (cube.getParticle(i).getPosition()[1] < position[1] + eEPSILON) {  //碰撞判定
            cube.getParticle(i).addForce(coll);
        }
    }
}
```

斜坡碰撞

```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;

    float radius = 3.0f;
    float mass = 10.0f;
    // TODO
    int par_num = cube.getParticleNum();

    Eigen::Vector3f coll;
    coll[0] = 4.9f * sqrt(2);
    coll[1] = 4.9f * sqrt(2);
    coll[2] = 0;

    for (int i = 0; i < 1000; i += 1) {
        if (cube.getParticle(i).getPosition()[1] + cube.getParticle(i).getPosition()[0] <
            position[1]  + eEPSILON + 1) {  //碰撞判定
            cube.getParticle(i).addForce(coll);
        }
    }


}
```

Result and Discussion

彈簧串聯結果如下:



最終結果如下列影片所示，結果並未很理想，在實作表面碰撞跟積分器的時候都有遇到一些問題，且跑起來非常的卡，不像範例影片裡面的流暢度，最後還是不確定要如何把積分器與程式裡的其他物件連結在一起運作，因此只能做出一個不完整的系統。

影片 1

影片 2

Conclusion

在這次的實作中讀懂了許多上課並未完全理解的部分，也靠著實作更加深了印象，雖然有許多部份沒有實作出來，幾乎只完成彈簧建構與彈簧阻尼力的計算部分，但仍然在嘗試其他的 function 時學到了很多，也充分了解到自己物理數學能力嚴重不足，之後需要再花更多時間去精熟這些科目。

最後也謝謝助教耐心的看完如此不完整的報告與結果。