Introduction to Computer Animation HW1 Report

資工系 0416303 楊博凱

- ## Introduction/Motivation

這次作業我們是用粒子系統模擬簡單的布，並且模擬球碰撞在布上面的情形。作業中粒子相互以彈簧連接，彼此之間存在內力(Spring Force and Damper Force)，相互構成一個網子，用來蒐集飛出的球。因此除了內力，我們也要計算網子(粒子)與地面的碰撞，以及網子與球的碰撞。而飛出的球則可以看成是一個巨大的粒子，故要處理球之間的碰撞以及球與地面的碰撞。

- ## Fundamentals

  - ### Internal Force

    彈簧的內力可以分成彈簧力與阻尼力，彈簧力主要功能是在振動發生時暫時儲存振動能量，而阻尼力的主要功能則是在振動發生時消減振動能量。因此在實作中，我們對於每個彈簧，都要計算他受到的彈簧力與阻尼力，加上現在受到的力，成為合力。

  - ### Collision

    - **Ball / Ball**

      球與球的碰撞可以看成兩個粒子之間的碰撞，透過計算半徑與相對距離以及相對速度關係，可以得知兩球是否接觸，進而算出兩球碰撞後的切線速度與法線速度。

    - **Ball / Net**

      球與網子的碰撞，實際上就是一個粒子與多個粒子的碰撞；與球球碰撞相同，透過計算半徑與相對距離以及相對速度關係，可以得知球與粒子是否有接觸，求出兩球碰撞後的切線速度與法線速度。搭配彈簧內力計算出粒子真正受到的力與速度。

    - **Ball / Plane**

      球與地面的碰撞事實上就是單一粒子與平面的碰撞，透過計算正向力與球位置與地面的關係以及正向力與球相對速度的關係，我們可以判斷球是否接觸到了地面。由於這次作業要我們考慮粒子撞到地面時具有摩擦力，因此除了考慮法線方向具有抵抗力，切線方現也要注意摩擦力。

    - **Net / Plane**

      網子與地面的碰撞可以看成多個粒子與地面的碰撞，而對每個粒子也可以分開計算其所受的力，最後在搭配內力算出合力。如上，透過計算正向力與球位置與地面的關係以及正向力與球相對速度的關係，我們可以判斷網子上的粒子是否接觸到了地面。當然我們一樣必須考慮摩擦力。

- Integration
  - **Explicit Euler Method**

    Euler Method 是常微分方程式的數值解法中較簡單的一種，效率不高、準確度也較差，但是相對好計算。他是運用現在的數值加上一個小區間內的變化量，也就是將小區間數值乘以數值的微分。

  - **Runge Kutta 4th Method**

    為了改善 Euler Method，我們在利用上面的方法計算下一步的數值時，先在求函數 f 的近似值並加以適當修正。考慮在 x(i) 至 x(i+1) 中選取幾個特定點，分別計算它的函數值，並加上適當的權重，最後用以計算正確的 f 值。

Implementation

- Initialize Spring

  我透過畫出實際上每個粒子在陣列中的 index，針對五個網子平面找出每個粒子之間 index 的規律，找出每個 Spring。因此實作中，我一共分成五個 Part 完成 Spring 的初始化。由於在 code 中找不到彈簧的 Restlength，因此我從 Configutation.txt 中找到網子的長度以及每個邊的粒子數，用長度除以(粒子數-1)，算出 Restlength。

  五個網子其中一面：

```
// TO DO 1
// Left Front
for (int id = 0; id < m_NumAtWidth * m_NumAtHeight; ++id){
    if ((id + 1) % m_NumAtWidth != 0){
        startParticleId = id;
        CSpring spring(startParticleId, startParticleId + 1, restLength, 1500.0, 50, color, (CSpring::enType_t) 0);
        m_Springs.push_back(spring);
    }
    if (id >= m_NumAtWidth){
        startParticleId = id;
        CSpring spring(startParticleId, startParticleId - 10, restLength, 1500.0, 50, color, (CSpring::enType_t) 0);
        m_Springs.push_back(spring);
    }
}
```

- Internal Force

  如果粒子位於比靜止位置更遠的位置，則彈簧力需要將其拉回；如果粒子位於比靜止位置更近的位置，則彈簧力需要將其推開。然而在公式上彈簧力推導出來都是一樣的。而阻尼力也是一樣的道理，只是為是阻止運動，使得粒子系統在沒有外力的情況下逐漸停止。

  合力只是將兩者加上現在所受到的力。

```
Vector3d GoalNet::ComputeSpringForce(
    const Vector3d &a_crPos1,
    const Vector3d &a_crPos2,
    const double a_cdSpringCoef,
    const double a_cdRestLength
    )
{
    //TO DO 3
    Vector3d force = (-a_cdSpringCoef) * ((a_crPos1 - a_crPos2).Length() - a_cdRestLength) * (a_crPos1 - a_crPos2) / (a_crPos1 - a_crPos2).Length();
    return force;
}

Vector3d GoalNet::ComputeDamperForce(
    const Vector3d &a_crPos1,
    const Vector3d &a_crPos2,
    const Vector3d &a_crVel1,
    const Vector3d &a_crVel2,
    const double a_cdDamperCoef
    )
{
    //TO DO 4
    Vector3d force = (-a_cdDamperCoef) * (a_crVel1 - a_crVel2).DotProduct(a_crPos1 - a_crPos2) / (a_crPos1 - a_crPos2).Length() * (a_crPos1 - a_crPos2) / (a_crPos1 - a_crPos2).Length();
    return force;
}
```

- Collision
  - 粒子與平面碰撞：(Ball / Plane；Net / Plane)

    針對每個粒子，判斷平面對他的正向力是否與粒子位置跟平面上的點形成的向量互相垂直，再判斷正向力與粒子相對速度向量的方向是否相互接近。若以上兩條件均滿足，則代表碰撞發生；針對切線方向，我們給他摩擦力，而法線方向，我們將速度反向並乘上抵抗係數。

```
for (int id = 0; id < m_GoalNet.ParticleNum(); id++){
    if (N.DotProduct(m_GoalNet.GetParticle(id).GetPosition() - ground) < eEPSILON && N.DotProduct(m_GoalNet.GetParticle(id).GetVelocity()) < 0){
        Vector3d newVelocity;
        newVelocity[1] = (-resistCoef) *m_GoalNet.GetParticle(id).GetVelocity()[1];
        newVelocity[0] = m_GoalNet.GetParticle(id).GetVelocity()[0];
        newVelocity[2] = m_GoalNet.GetParticle(id).GetVelocity()[2];
        m_GoalNet.GetParticle(id).SetVelocity(newVelocity);

        Vector3d frictionForce;
        Vector3d vt(m_GoalNet.GetParticle(id).GetVelocity()[0], 0, m_GoalNet.GetParticle(id).GetVelocity()[2]);
        frictionForce = (-frictionCoef) * (-N.DotProduct(m_GoalNet.GetParticle(id).GetForce())) * vt;
        m_GoalNet.GetParticle(id).AddForce(-frictionForce);
    }
    if (m_GoalNet.GetParticle(id).GetPosition()[1] < -1){
        Vector3d tempP(m_GoalNet.GetParticle(id).GetPosition()[0], 0, m_GoalNet.GetParticle(id).GetPosition()[2]);
        m_GoalNet.GetParticle(id).SetPosition(tempP);
    }
}
```

```
for (int id = 0; id < m_Balls.size(); id++){

    if (N.DotProduct(m_Balls[id].GetPosition() - ground) < eEPSILON && N.DotProduct(m_Balls[id].GetVelocity()) < 0){
        Vector3d newVelocity;
        newVelocity[1] = (-resistCoef) * m_Balls[id].GetVelocity()[1];
        newVelocity[0] = m_Balls[id].GetVelocity()[0];
        newVelocity[2] = m_Balls[id].GetVelocity()[2];
        m_Balls[id].SetVelocity(newVelocity);

        Vector3d frictionForce;
        Vector3d vt(m_Balls[id].GetVelocity()[0], 0, m_Balls[id].GetVelocity()[2]);
        frictionForce = (-frictionCoef) * (-N.DotProduct(m_Balls[id].GetForce())) * vt;
        m_Balls[id].AddForce(-frictionForce);
    }
}
```

  - 粒子與粒子碰撞：(Ball / Ball；Ball / Net)

    針對每個粒子我們判斷他們彼此之間的關係有兩項：

    1. 兩者之間的距離是否小於其半徑相加
    2. 兩粒子是否相互接近中 (兩粒子中固定一粒子，令一粒子的相對速度與位置向量內積是否小於零)

    若以上條件滿足則代表碰撞發生，針對切線速度及法線速度做計算，並針對球碰撞到竿子(固定的粒子)另外做處理。

```
for (int id = 0; id < m_Balls.size(); id++){
    for (int j = 0; j < m_Balls.size(); j++){
        if (id != j && (m_Balls[id].GetPosition() - m_Balls[j].GetPosition()).Length() < (m_Balls[id].GetRadius() + m_Balls[j].GetRadius()) && (m_Balls[j].GetPosition() - m_Balls[id].GetPosition()).DotProduct(m_Balls[j].GetVelocity() - m_Balls[id].GetVelocity()) < 0){
            Vector3d particleToBall = m_Balls[id].GetPosition() - m_Balls[j].GetPosition();
            Vector3d vln = (particleToBall.DotProduct(m_Balls[j].GetVelocity()) / (particleToBall.Length() * particleToBall.Length())) * particleToBall;
            Vector3d vlt = m_Balls[j].GetVelocity() - vln;
            Vector3d ballToParticle = -particleToBall;
            Vector3d v2n = (ballToParticle.DotProduct(m_Balls[id].GetVelocity()) / (ballToParticle.Length() * ballToParticle.Length())) * ballToParticle;
            Vector3d v2t = m_Balls[id].GetVelocity() - v2n;
            m_Balls[j].SetVelocity((vln * (m_Balls[j].GetMass() - m_Balls[id].GetMass()) + 2 * m_Balls[id].GetMass() * v2n) / (m_Balls[j].GetMass() + m_Balls[id].GetMass()) + vlt);
            m_Balls[id].SetVelocity((v2n * (m_Balls[id].GetMass() - m_Balls[j].GetMass()) + 2 * m_Balls[j].GetMass() * vln) / (m_Balls[j].GetMass() + m_Balls[id].GetMass()) + v2t);
        }
    }
}
```

```
for (int id = 0; id < m_Balls.size(); id++){
    for (int j = 0; j < m_GoalNet.ParticleNum(); j++){
        if (m_GoalNet.GetParticle(j).IsMovable()){
            if ((m_Balls[id].GetPosition() - m_GoalNet.GetParticle(j).GetPosition()).Length() < m_Balls[id].GetRadius() && (m_GoalNet.GetParticle(j).GetPosition() - m_Balls[id].GetPosition()).DotProduct(m_GoalNet.GetParticle(j).GetVelocity() - m_Balls[id].GetVelocity()) < 0){
                Vector3d particleToBall = m_Balls[id].GetPosition() - m_GoalNet.GetParticle(j).GetPosition();
                Vector3d v1n = (particleToBall.DotProduct(m_GoalNet.GetParticle(j).GetVelocity()) / (particleToBall.Length() * particleToBall.Length())) * particleToBall;
                Vector3d v1t = m_GoalNet.GetParticle(j).GetVelocity() - v1n;
                Vector3d ballToParticle = -particleToBall;
                Vector3d v2n = (ballToParticle.DotProduct(m_Balls[id].GetVelocity()) / (ballToParticle.Length() * ballToParticle.Length())) * ballToParticle;
                Vector3d v2t = m_Balls[id].GetVelocity() - v2n;
                m_GoalNet.GetParticle(j).SetVelocity((v1n * (m_GoalNet.GetParticle(j).GetMass() - m_Balls[id].GetMass()) + 2 * m_Balls[id].GetMass() * v2n) / (m_GoalNet.GetParticle(j).GetMass() + m_Balls[id].GetMass()) + v1t);
                m_Balls[id].SetVelocity((v2n * (m_Balls[id].GetMass() - m_GoalNet.GetParticle(j).GetMass()) + 2 * m_GoalNet.GetParticle(j).GetMass() * v1n) / (m_GoalNet.GetParticle(j).GetMass() + m_Balls[id].GetMass()) + v2t);
            }
        }
        else{
            if ((m_Balls[id].GetPosition() - m_GoalNet.GetParticle(j).GetPosition()).Length() < m_Balls[id].GetRadius() && (m_GoalNet.GetParticle(j).GetPosition() - m_Balls[id].GetPosition()).DotProduct(m_GoalNet.GetParticle(j).GetVelocity() - m_Balls[id].GetVelocity()) < 0){
                Vector3d particleToBall = m_Balls[id].GetPosition() - m_GoalNet.GetParticle(j).GetPosition();
                Vector3d v1n = (particleToBall.DotProduct(m_GoalNet.GetParticle(j).GetVelocity()) / (particleToBall.Length() * particleToBall.Length())) * particleToBall;
                Vector3d v1t = m_GoalNet.GetParticle(j).GetVelocity() - v1n;
                Vector3d ballToParticle = -particleToBall;
                Vector3d v2n = (ballToParticle.DotProduct(m_Balls[id].GetVelocity()) / (ballToParticle.Length() * ballToParticle.Length())) * ballToParticle;
                Vector3d v2t = m_Balls[id].GetVelocity() - v2n;
                m_Balls[id].SetVelocity(-v2n + v2t);
            }
        }
    }
}
```

- Integration
  - Euler Method

    針對每個粒子計算下一時間點的位置以及速度。取適當 deltaT = 0.001，乘以變化量，即為微分值，再加上原本的數值，則是 deltaT 時間後的解。

```
// Goal Net
for (int pIdx = 0; pIdx < m_GoalNet.ParticleNum(); ++pIdx)
{
    //TO DO 6
    double deltaT = 0.001;
    m_GoalNet.GetParticle(pIdx).SetVelocity(m_GoalNet.GetParticle(pIdx).GetVelocity() + deltaT * m_GoalNet.GetParticle(pIdx).GetAcceleration());
    m_GoalNet.GetParticle(pIdx).SetPosition(m_GoalNet.GetParticle(pIdx).GetPosition() + deltaT * m_GoalNet.GetParticle(pIdx).GetVelocity());

}

// Balls
for (int ballIdx = 0; ballIdx < BallNum(); ++ballIdx)
{
    //TO DO 6
    double deltaT = 0.001;

    m_Balls[ballIdx].SetVelocity(m_Balls[ballIdx].GetVelocity() + deltaT * m_Balls[ballIdx].GetAcceleration());
    m_Balls[ballIdx].SetPosition(m_Balls[ballIdx].GetPosition() + deltaT * m_Balls[ballIdx].GetVelocity());
}
```

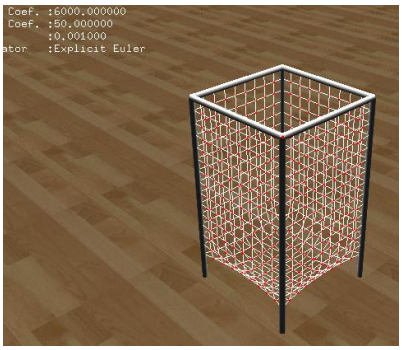
  - Runge Kutta 4th

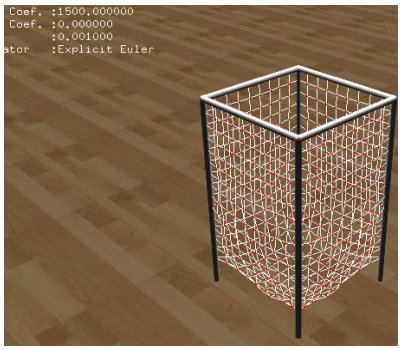
    此方法是讓欲求的數值由更多不同樣式的半步組合所構成，其中並包含了起點斜率與終點的斜率。其中 k1 是時間開始時函數的斜率；k2 是時間中間的斜率，通過 Euler Method 採用斜率 k1 來決定 y 在點 t(n) + h/2 的值；k3 也是中點的斜率，但是採用斜率 k2 決定 y 值；k4 是時間段終點的斜率，其 y 值用 k3 決定。

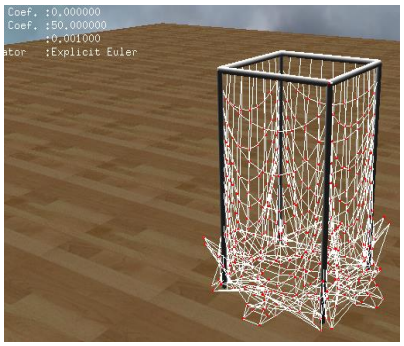
```
double h = 0.001;
for (int id = 0; id < m_GoalNet.ParticleNum(); id++){

    struct StateStep tempState1;
    Vector3d posk1 = m_GoalNet.GetParticle(id).GetPosition();
    Vector3d velk1 = m_GoalNet.GetParticle(id).GetVelocity();
    Vector3d acc = m_GoalNet.GetParticle(id).GetAcceleration();
    tempState1.deltaPos = posk1;
    tempState1.deltaVel = velk1;
    k1StepCntr.push_back(tempState1);

    struct StateStep tempState2;
    Vector3d posk2 = m_GoalNet.GetParticle(id).GetPosition() + 0.5 * k1StepCntr[id].deltaVel * h;
    Vector3d velk2 = m_GoalNet.GetParticle(id).GetVelocity() + 0.5 * acc * h;
    tempState2.deltaPos = posk2;
    tempState2.deltaVel = velk2;
    k2StepCntr.push_back(tempState2);

    struct StateStep tempState3;
    Vector3d posk3 = m_GoalNet.GetParticle(id).GetPosition() + 0.5 * k2StepCntr[id].deltaVel * h;
    Vector3d velk3 = m_GoalNet.GetParticle(id).GetVelocity() + 0.5 * acc * h;
    tempState3.deltaPos = posk3;
    tempState3.deltaVel = velk3;
    k3StepCntr.push_back(tempState3);

    struct StateStep tempState4;
    Vector3d posk4 = m_GoalNet.GetParticle(id).GetPosition() + k3StepCntr[id].deltaVel * h;
    Vector3d velk4 = m_GoalNet.GetParticle(id).GetVelocity() + acc * h;
    tempState4.deltaPos = posk4;
    tempState4.deltaVel = velk4;
    k4StepCntr.push_back(tempState4);

    m_GoalNet.GetParticle(id).AddPosition((h / 6.0) * (k1StepCntr[id].deltaVel + 2 * k2StepCntr[id].deltaVel + 2 * k3StepCntr[id].deltaVel + k4StepCntr[id].deltaVel));
    m_GoalNet.GetParticle(id).AddVelocity((h / 6.0) * (acc + 2 * acc + 2 * acc + acc));
}
```

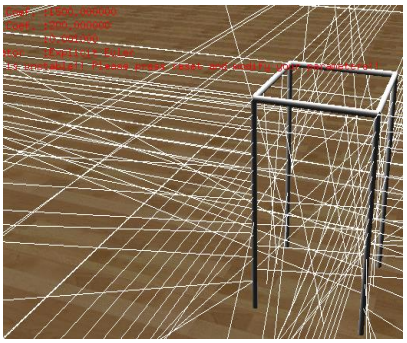
- Result and Discussion
  - The difference between Explicit Euler and RK4

    Runge Kutta Method 可以看作是 Explicit Euler 在中間值的多種應用；因 Explicit Euler 為一階一階慢慢計算，而 Runge Kutta Method 一次走很多階然後算出一點代表這幾階。通常 Explicit Euler 的誤差比 Runge Kutta Method 高，因為與 Euler Method 相比，Runge Kutta Method 中的 truncation error 較小。
  - Effect of parameters
    1. 彈簧係數若越低，阻尼係數若越高，則網子無法穩定，而會導致按下 start 後 crash。倘若彈簧係數若越高，阻尼係數若越低，則不會發生問題。









    2. 若計算 integration 時 deltaT 太大，則容易產生誤差；若 deltaT 太小，則容易產生 overflow 的問題，或是導致計算量太高。
    3. 彈簧 Restlength 若太短則彈簧會太緊繃，而按下 start 後網子碰不到地面；若 restlength 太長，則網子太多部分會垂到地面，而球則會直接穿過網子。

- Conclusion
  - 觀察 FPS 後發現當 Integration 方法為 Euler 時 FPS 較大，而方法為 Runge Kutta 時 FPS 降得較低，可見 Runge Kutta 的計算量確實大於 Euler。然而精準度由於肉眼無法清楚辨識，因此不能證明 Runge Kutta 的誤差較 Euler 小。
  - 由於我們沒有做球與彈簧的碰撞，因此當球掉進網子中間時，會發生彈簧陷入球的狀況。
  - 參數的數值要慎選，deltaT 不可太大也不可太小；抵抗係數、彈簧係數、阻尼係數均會影響到系統的穩定程度。
  - 粒子即使沒有受到球的碰撞，再接觸地面時，也會自己產生些許震動情況，存在不穩定性。而當粒子數量變多，則模擬的情況越接近真實的布料；若將時間間隔 deltaT 調小，則不穩定性較難被發現。
  - 若不做球與竿子的碰撞，則當球打到網子邊緣 (即為竿子所在)時，球會黏在上面，甚至是陷進去。