

# Computer Animation and Special Effects

## Hw1 Particle System Report

310551067 吳岱容

- **Introduction**

此次作業主要目的是利用課堂所學做出以粒子系統模擬球體間碰撞以及球體與布料間的碰撞現象，主要實作內容可以分成三個部分：

- 以彈簧系統構築布料結構
- 碰撞處理
- 積分器(Explicit, Implicit, Midpoint, Runge Kutta Fourth)

- **Fundamental**

- **Spring System**

透過粒子間連結的彈簧來模擬布料的型態，其中構成的彈簧可以分為三種(structure, shear, bend)用以來推動粒子。

$$\mathbf{f}_a = -k_s(|\mathbf{x}_a - \mathbf{x}_b| - l_0) \frac{\mathbf{x}_a - \mathbf{x}_b}{|\mathbf{x}_a - \mathbf{x}_b|}$$
$$-k_d \left( \frac{(\mathbf{v}_a - \mathbf{v}_b) \cdot (\mathbf{x}_a - \mathbf{x}_b)}{|\mathbf{x}_a - \mathbf{x}_b|} \right) \frac{\mathbf{x}_a - \mathbf{x}_b}{|\mathbf{x}_a - \mathbf{x}_b|}$$

彈簧的力可以分為兩種：彈簧力(spring force)以及阻尼力(damping force)，其中彈簧力來自於虎克定律，會與彈性係數以及彈簧壓縮量有關，阻尼力則是由於負功使系統逐漸趨於停止的力(可能來自阻力、摩擦力)，會與阻尼係數以及物體速度相關。

- **Collision**

碰撞的實作主要可以分為三個部分：偵測碰撞、防止穿模、處理碰撞，處理碰撞部分又可以分為球體間的碰撞以及球體與布料間的碰撞，因為我們實作的是由粒子系統模擬，那每一個碰撞其實都可以看做質點間的碰撞處理，透過改變質點速度來達成。

$$v_a = \frac{m_a u_a + m_b u_b + m_b C_R (u_b - u_a)}{m_a + m_b}$$

and

$$v_b = \frac{m_a u_a + m_b u_b + m_a C_R (u_a - u_b)}{m_a + m_b}$$

- **Integrator**

積分器在此的作用是以現有的系統狀態來更新物體的速度以及位置，在此次的作業我們一共實現了4種積分器(Explicit, Implicit, Midpoint, Runge Kutta Fourth)。

在Explicit Method中，我們取現在的點資訊來進行下一步的更新：

$$\mathbf{x}(t + h) = \mathbf{x}(t) + h \cdot f(\mathbf{x}, t)$$

它是最簡單的方法，但同時也會隨著時間取樣區間的擴大而出現誤差擴大的情況。

Implicit Method則是透過下一個時間點的點資訊來進行下一步更新：

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

雖然我們必須先去估計在下一情況的資訊點，但較於Explicit，這是比較穩定的做法。

Midpoint Method的取樣則是在中點：

$$\mathbf{x}(t_0 + h) = \mathbf{x}(t_0) + h \cdot f_{mid}$$

Runge Kutta則是透過四個點做加權估計：

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

$$k_1 = hf(\mathbf{x}_0, t_0)$$

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

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

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

- **Implementation**

- **void Cloth::initializeSpring()**

在這邊我們主要是遍歷每一個布料的粒子，然後連結他們之間的彈簧，共有三種類型，各自以不同的迴圈來建立，structure除邊界粒子每個粒子的上下左右皆會連接一條structure spring，shear則是以對角線方式連接，bending則是跨越一個粒子連接。

- **void Cloth::computeSpringForce()**

計算彈簧給予粒子的作用力，將其換算成對質點的加速度，主要藉由虎克定律來計算。

```
auto distAB = (_particles.position(aIdx) - _particles.position(bIdx)).norm();
auto forceVec = (_particles.position(aIdx) - _particles.position(bIdx)).normalized();
```

此計算需要質點間的距離以及之間的方向向量(力作用方向)，即可藉由虎克定律來算出彈簧力。

```
auto springF = -springCoef * (distAB - spring.length()) * forceVec - damperCoef * vDotL * forceVec
```

最後再將其換成加速度儲存。

```
_particles.acceleration(aIdx) = _particles.acceleration(aIdx) + springF * _particles.inverseMass(aIdx);
_particles.acceleration(bIdx) = _particles.acceleration(bIdx) - springF * _particles.inverseMass(bIdx);
```

- **void Spheres::collide(Cloth\* cloth)**

此處計算的是球體與布料間的碰撞處理。

首先是碰撞的偵測，布料的粒子我們假設為質點，而球體則有半徑，所以當兩質點距離為球體半徑時將偵測為碰撞成立。

```
//collision solution
if (dist < this->radius(k)) {
```

在某些時候運算上可能會導致穿模，在此這裡使物體移動小距離來避免此種問題發生。

```
auto correction = (this->radius(k) - dist) * normal * 0.15f;
cloth->particles().position(i * particlesPerEdge + j) += correction;
this->particles().position(k) -= correction;
```

再來則是碰撞的處理，碰撞的兩物體我們藉由改變其速度來達成效果。

```
this->particles().velocity(k) =
(ballMass * ballV + clothMass * clothV + coefRestitution * clothMass * (clothV - ballV)) /
(ballMass + clothMass);
cloth->particles().velocity(i * particlesPerEdge + j) =
(clothMass * clothV + ballMass * ballV + coefRestitution * ballMass * (ballV - clothV)) /
(ballMass + clothMass);
```

- **void Spheres::collide()**

這邊處理的是球體間的碰撞，做法與上述衣料碰撞類似，只是球體非質點，因次碰撞條件必須改為兩個球體半徑。

- **void ExplicitEuler::integrate(...)**

此處實作的是Explicit方法，就是用deltaTime以及現在的點資訊來更新速度以及位置情況。

```
for (const auto &p : particles) {
    // Write code here!
    p->velocity() = p->velocity() + p->acceleration() * deltaTime;
    p->position() = p->position() + p->velocity() * deltaTime;
}
```

- **void ImplicitEuler::integrate(...)**

在Implicit方法中，我們必須先去模擬未來下一時間點的資訊，再以此資訊

更新現在的粒子。

```
// update
p->velocity() = p->velocity() + p->acceleration() * deltaTime;
p->position() = p->position() + p->velocity() * deltaTime;
simulateOneStep();
p->velocity() = curV+p->acceleration() * deltaTime;
p->position() = curPos + p->velocity() * deltaTime;
```

- **void MidpointEuler::integrate(...)**

Midpoint的做法和Implicit相似，只是這次的deltaTime只有二分之一。

```
//half step
p->velocity() += p->acceleration() * deltaTime / 2;
p->position() += p->velocity() * deltaTime / 2;
```

- **void RungeKuttaFourth::integrate(...)**

在Runge Kutta Fourth方法中，我們必須用類似的方法取得四點的估計值。

```
// k1
auto k1V = curV + p->acceleration() * deltaTime;
auto k1Pos = k1V * deltaTime;
// k2
p->velocity() += p->acceleration() * deltaTime / 2;
p->position() += k1Pos / 2;
simulateOneStep();
auto k2V = curV + p->acceleration() * deltaTime;
auto k2Pos = k2V * deltaTime;
p->velocity() = curV;
p->position() = curPos;

// k3
p->velocity() += p->acceleration() * deltaTime / 2;
p->position() += k2Pos / 2;
simulateOneStep();
auto k3V = curV + p->acceleration()*deltaTime;
auto k3Pos = k3V * deltaTime;
p->velocity() = curV;
p->position() = curPos;
//k4
p->velocity() += p->acceleration() * deltaTime;
p->position() += k3Pos;
simulateOneStep();
auto k4V = curV + p->acceleration() * deltaTime;
auto k4Pos = k4V * deltaTime;
```

最後再將其更新。

```
// update
auto a = curPos + (k1Pos + 2 * k2Pos + 2 * k3Pos + k4Pos) / 6;
p->velocity() = curV + p->acceleration() * deltaTime;
p->position() = a;
```

- **Result and Discussion**

- **Explicit Method**

這個方法是最簡單實作的，相較於其他方式可以維持在比較穩定的fps（通常為60fps），雖然在時間取樣短的情況看不太出來，但較於其他方法來說動得比較僵硬，而且time stamp在約0.00023的情況下就會出現崩潰的情形，是4種方法裡面最低的。

- **Implicit Method**

在同樣的deltaTime下，比起Explicit更吃效能（同樣為0.0001情況下fps會下降到約30），但是在長deltaTime情況下有比較好的表現也比Explicit不容易崩潰（可以撐到約0.00025左右）。

- **Midpoint Method**

在效能下與Implicit差不多，但視覺效果上比起Implicit顯得要平均一點，而在長deltaTime有比起Implicit更好的耐受程度，可以撐到約0.00035還不崩潰。

- **Runge Kutta Fourth Method**

同樣deltaTime下是最吃效能的方法（在0.0001的情況下fps會降到30以下），但是同時也是可以耐受最長deltaTime的方法（約在0.00036才會崩潰），在長deltaTime的情況下也能維持不錯的模擬效果，這些結果的原因可能來自於多個點的取樣來進行修正，但同時多點的估算也會造成效能的損失。

- **Spring Coefficient**

基本上Spring coefficient會影響到彈簧拉伸的困難度，如果越小彈簧越容易被拉伸，布料會越容易被拉長，同時也越難回復，如果越大則反之。

- **Damping Coefficient**

Damping Coefficient代表的則是系統的穩定性，如果過大會發生彈簧幾乎不振動的情況，過小則會發生極不穩定、一直抖動的情況。

- **Conclusion**

此次作業中我們主要實作粒子系統來模擬布料以及碰撞，透過這次作業，對於上課所學有更進一步的認識，至少透過實作，相關的知識也更加具現化。

作業中我們做了多種的不同的積分器，從此也可以看得出來每種方法都有其各自的缺點，也有各自的使用時機。