國立交通大學電機資訊學士班資訊書報專題

專題名稱: 電腦動畫原理探究與製作
指導教授:林文杰 教授
學生:0710019 黃煒恩 0710026 薛立甜

Soft Body Simulation

Motivation

幾年前,迪士尼推出一部動畫帶來一股席捲全球的潮流－冰雪奇緣(Frozen)。當時看完這部動畫時,我們不只被優美的配樂以及創新的劇情吸引,更被那冰雪紛飛、銀白雪國的場景深深的震懾住。為了模擬出最為真實的雪景,冰雪奇緣團隊找來許多專家一同考察,據說光是一個城堡鏡頭,就要 50 位動畫師耗時 30 小時製作。因此我們想藉著這次專題的製作,一窺電腦動畫的奧妙,希望能實際在電腦中模擬出一個動畫的小小天地。

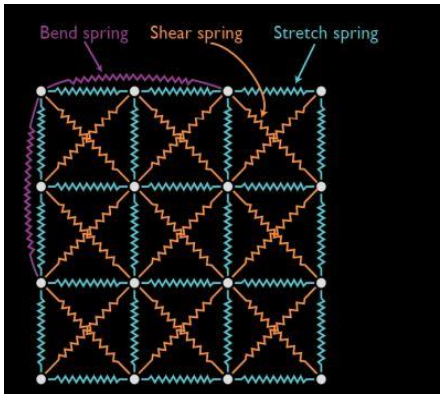
Introduction

在 Soft Body Simulation 中,我們模擬一顆軟骰子因重力落下,並與地形發生碰撞、反彈的情形。那首先,我們該如何模擬出一顆軟軟的骰子呢?我們創造一個正立方體,內含 1000 個粒子(particle),並在粒子之間適當的加上一些彈簧來模擬粒子與粒子之間的連結,藉由每個小小的內部彈簧壓縮伸展和阻尼減緩震盪來模擬整個軟骰子從受力的瞬間、凹陷、回彈到最終回歸靜止過程裡每個瞬間每個粒子的受力情況、加速度、速度和位置。除此之外,也模擬平面、斜坡、碗和球等四種不同地形對於掉落的骰子的碰撞情況。

模擬主要分為下列幾個部分:

• Construct the connection of springs(Spring Initialization)
• Compute spring and damper forces
• Handle collision
- Plane and Tilted Plane
- Sphere
- Bowl
- Contact force (resist and friction)
• Integrator
- Explicit Euler
- Implicit and Midpoint Euler
- Runge Kutta 4th

Spring Initialization



對於一個由粒子組成的 soft body cube 會有三種 spring 連接 particles，分別為連接相鄰粒子的 stretch spring、連接斜向粒子的 shear spring 及跨一個粒子連接的 bend spring 如右圖所示。

以 10^3 共 1000 粒子的 cube 為例，共有 2700 stretch springs + 7776 shear springs + 2400 bend springs.

Spring and Damper Forces

- Spring Forces

若兩粒子距離大於彈簧靜止長度(rest length)，彈簧力需要將其拉回；若兩粒子距離小於彈簧靜止長度，則彈簧力需要將其推開。

施加在粒子 a 的彈簧力：

$$\vec{f}_a = -k_s(|\vec{x}_a - \vec{x}_b| - r)\vec{l}, \quad k_s > 0 \quad \vec{l} = \frac{\vec{x}_a - \vec{x}_b}{|\vec{x}_a - \vec{x}_b|}$$

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

r: rest length

k_s: spring coefficient

- Damper Forces 阻尼力

如果兩個粒子分離，則阻尼力需要將它們拉回；如果兩個顆粒接近，則阻尼力需要將它們推開。

施加在粒子 a 的阻尼力：

$$\vec{f}_a = -k_d((\vec{v}_a - \vec{v}_b)\cdot\vec{l})\vec{l}, \quad k_d > 0 \quad \vec{l} = \frac{\vec{x}_a - \vec{x}_b}{|\vec{x}_a - \vec{x}_b|}$$

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

k_d : damper coefficient

- Effect of coefficient

當 spring coefficient 越大，粒子受彈簧力作用越明顯；而當 damper coefficient 越大，物體產生形變時，越容易復原。實作時我們發現，damper coefficient 最

好不要設成 0，這樣比較接近真實狀態；而當 damper coefficient 調太大則會造成系統不穩定。

Handle collision

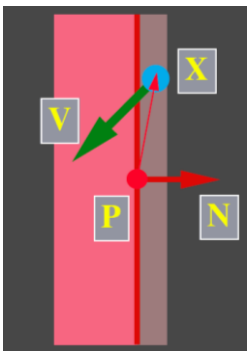
處理不同地形的碰撞情形，並計算碰撞時的地面給予的接觸力和摩擦力。

- 碰撞的偵測:
  1. 粒子距離地面夠近
  2. 粒子速度內積地面法向量(N)小於零

  符合上述條件判斷為碰撞。

- Plane and Tilted Plane

1. $\mathbf{N}\cdot(\mathbf{x}-\mathbf{p})<\varepsilon$
2. $\mathbf{N}\cdot\mathbf{v}<0$



- Sphere

1.若粒子距離圓心<=radius + ε

2. $\mathbf{N}\cdot\mathbf{v}<0$

- Bowl

分成物體在碗外/內討論，碗外偵測方式與球體相同

碗內:

1.若粒子距離圓心>=radius - ε

2. $\mathbf{N}\cdot\mathbf{v}<0$

- 碰撞後速度:

Before: $\mathbf{v}=\mathbf{v}_N+\mathbf{v}_T$ (將 v 分成與法向量平行的 $\mathbf{v}_N$ 以及 $\mathbf{v}_T$)

After: $\mathbf{v}'=-k_r\ \mathbf{v}_N+\mathbf{v}_T$ k_r :coefficient of restitution

- 碰撞時受力:

當 $\mathbf{N}\cdot\mathbf{f}<0$

粒子所受接觸力 $\mathbf{f}^c=-(\mathbf{N}\cdot\mathbf{f})\,\mathbf{N}$

粒子所受摩擦力 $\mathbf{f}^f = -k_f(-\mathbf{N}\cdot\mathbf{f})\,v_t$　k_f :摩擦係數

當 $\mathbf{N}\cdot\mathbf{f} < 0$，$\mathbf{f}^c = \mathbf{f}^f = 0$

Integrator

設微分方程　$y'(t) = f(t, y(t))$，並且 t_n 和 t_{n+1} 時間差 $t_{n+1} - t_n = \Delta t$

Explicit Euler's method 顯性歐拉法

顯性歐拉積分定義為 $y_{n+1} = y_n + \Delta t \cdot f(t_n, y_n)$

亦即以 t 時刻的斜率作整段過程 t→t+Δt 的導數(斜率)，誤差項為 $O(\Delta t^2)$。

Drawbacks: 對於曲線近似不準確(僅考量初值、起點微分值和步長)、誤差累積大(迭代)。

```
//
// ExplicitEulerIntegrator
//
ExplicitEulerIntegrator::ExplicitEulerIntegrator() {}

IntegratorType ExplicitEulerIntegrator::Type()
{
    return IntegratorType::ExplicitEuler;
}

void ExplicitEulerIntegrator::Integrate(MassSpringSystem& particleSystem)
{
    double dt = particleSystem.deltaTime;
    for (int i = 0; i < particleSystem.cubeCount; i++)
    {
        int PartNum = particleSystem.cubes[i].ParticleNum();
        for (int j = 0; j < PartNum; j++)
        {
            particleSystem.cubes[i].GetParticle(j).AddVelocity(dt*particleSystem.cubes[i].GetParticle(j).Acceleration(
            particleSystem.cubes[i].GetParticle(j).AddPosition(dt*particleSystem.cubes[i].GetParticle(j).Velocity());
        }
    }
}
```

Midpoint method 中點法 (modified Euler's method 改良歐拉法)

設微分方程　$y'(t) = f(t, y(t))$

並且 t_n 和 t_{n+1} 時間差　$t_{n+1} - t_n = \Delta t$ ，

則中點積分(顯性)定義為　$y_{n+1} = y_n + \Delta t \cdot f(t_n + \frac{1}{2}\Delta t, y_n + \frac{1}{2}\Delta t \cdot f(t_n, y_n))$

隱性　$y_{n+1} = y_n + \Delta t \cdot f(tn + 1/2\Delta t, 1/2(y_n + y_{n+1}))$

亦即以中點斜率作為整個過程 t→t+Δt 的導數，誤差項為 $O(\Delta t^3)$。

```
//
// MidpointEulerIntegrator
//
MidpointEulerIntegrator::MidpointEulerIntegrator() {}
IntegratorType MidpointEulerIntegrator::Type()
{
    return IntegratorType::MidpointEuler;
}
void MidpointEulerIntegrator::Integrate(MassSpringSystem& particleSystem)
{
    particleSystem.deltaTime /= 2;
    for (int i = 0; i < particleSystem.cubeCount; i++)
    {
        int PartNum = particleSystem.cubes[i].ParticleNum();
        for (int j = 0; j < PartNum; j++)
        {
            //Taylor approach
            particleSystem.cubes[i].GetParticle(j).AddVelocity(particleSystem.deltaTime*particleSystem.cubes[i].GetParticle(j).Acceleration(
            particleSystem.cubes[i].GetParticle(j).AddPosition(particleSystem.deltaTime*particleSystem.cubes[i].GetParticle(j).Velocity());
        }
    }
    particleSystem.deltaTime *= 2;
}
```

Implicit Euler's method 隱性歐拉法

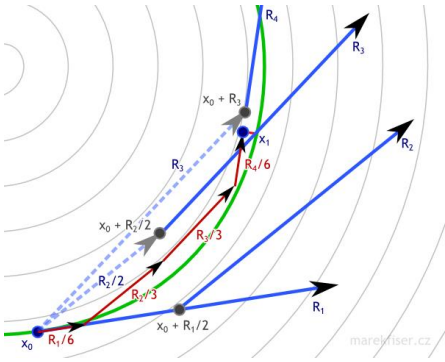
隱性歐拉積分定義為$y_{n+1} = y_n + \Delta t \cdot f(t_{n+1}, y_{n+1})$

亦即以 t 時刻的斜率作整段過程 t→t+Δt 的導數(斜率)，誤差項為$O(\Delta t^2)$。

```
//
// ImplicitEulerIntegrator
//
ImplicitEulerIntegrator::ImplicitEulerIntegrator() {}
IntegratorType ImplicitEulerIntegrator::Type()
{
    return IntegratorType::ImplicitEuler;
}

void ImplicitEulerIntegrator::Integrate(MassSpringSystem& particleSystem)
{
    double dt = particleSystem.deltaTime;
    for (int i = 0; i < particleSystem.cubeCount; i++)
    {
        int PartNum = particleSystem.cubes[i].ParticleNum();
        std::vector<Eigen::Vector3d> PrevPos, PrevVel;
        for (int j = 0; j < PartNum; j++) {
            PrevVel.push_back(particleSystem.cubes[i].GetParticle(j).Velocity());
            PrevPos.push_back(particleSystem.cubes[i].GetParticle(j).Position());
            particleSystem.cubes[i].GetParticle(j).SetForce(Eigen::Vector3d::Zero());
        }
        particleSystem.ComputeCubeForce(particleSystem.cubes[i]);
        for (int j = 0; j < PartNum; j++)
        {
            particleSystem.cubes[i].GetParticle(j).SetVelocity(PrevVel[j]+dt*particleSystem.cubes[i].GetParticle(j).Acceleration());
            particleSystem.cubes[i].GetParticle(j).SetPosition(PrevPos[j]+dt*particleSystem.cubes[i].GetParticle(j).Velocity());
        }
    }
}
```

Runge-Kutta 4 四階龍格-庫塔積分法

RK4 定義為$y_{n+1} = y_n + \Delta t \cdot 6 \cdot (k_1 + 2k_2 + 2k_3 + k_4)$



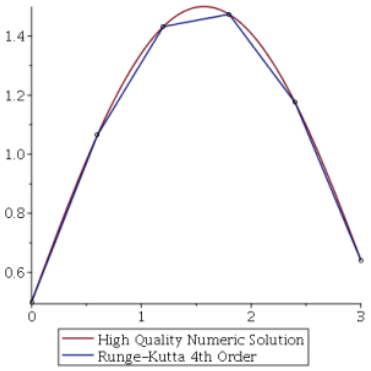
(右圖中 X_0 為 Y_n，R_i 即為 K_i 值，i = 1 , 2 , 3 , 4) 其中

$$k_1 = \mathrm{f}(t_n, y_n)$$

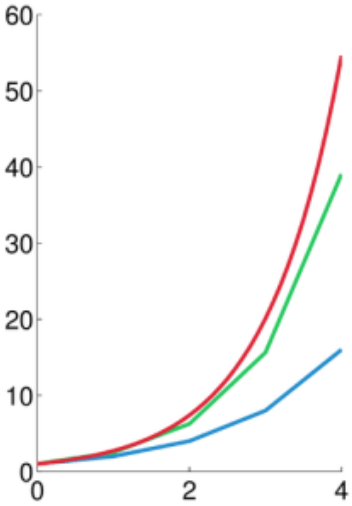
$$k_2 = \mathrm{f}(t_n + \frac{\Delta t}{2}, y_n + \frac{\Delta t}{2} \cdot k_2)$$

$$k_3 = \mathrm{f}(t_n + \frac{\Delta t}{2}, y_n + \frac{\Delta t}{2} \cdot k_2)$$

$$k_4 = \mathrm{f}(t_n + \Delta t, y_n + \Delta t \cdot k_3)$$




```
//
// RungeKuttaFourthIntegrator
//
RungeKuttaFourthIntegrator::RungeKuttaFourthIntegrator() {}
IntegratorType RungeKuttaFourthIntegrator::Type()
{
    return IntegratorType::RungeKuttaFourth;
}
void RungeKuttaFourthIntegrator::Integrate(MassSpringSystem& particleSystem)
{
    double dt = particleSystem.deltaTime;
    for (int i = 0; i < particleSystem.cubeCount; i++)
    {
        int PartNum = particleSystem.cubes[i].ParticleNum();
        std::vector<Eigen::Vector3d> curV, curX, k1x, k1v, k2x, k2v, k3x, k3v, k4x, k4v;
        for (int j = 0; j < PartNum; j++)
        {
            curV.push_back(particleSystem.cubes[i].GetParticle(j).Velocity());
            curX.push_back(particleSystem.cubes[i].GetParticle(j).Position());
            particleSystem.cubes[i].GetParticle(j).SetForce(Eigen::Vector3d::Zero());
        }

        particleSystem.ComputeCubeForce(particleSystem.cubes[i]);


        for (int j = 0; j < PartNum; j++)
        {
            k1v.push_back(particleSystem.deltaTime*particleSystem.cubes[i].GetParticle(j).Acceleration());
            k1x.push_back(particleSystem.deltaTime*particleSystem.cubes[i].GetParticle(j).Velocity());
            particleSystem.cubes[i].GetParticle(j).SetVelocity(curV[j] + 0.5*k1v[j]);
            particleSystem.cubes[i].GetParticle(j).SetPosition(curX[j] + 0.5*k1x[j]);
            particleSystem.cubes[i].GetParticle(j).SetForce(Eigen::Vector3d::Zero());
        }

        particleSystem.deltaTime /= 2;
        particleSystem.ComputeCubeForce(particleSystem.cubes[i]);
        particleSystem.deltaTime *= 2;

        for (int j = 0; j < PartNum; j++)
        {
            k2v.push_back(particleSystem.deltaTime*particleSystem.cubes[i].GetParticle(j).Acceleration());
            k2x.push_back(particleSystem.deltaTime*particleSystem.cubes[i].GetParticle(j).Velocity());
            particleSystem.cubes[i].GetParticle(j).SetVelocity(curV[j] + 0.5*k2v[j]);
            particleSystem.cubes[i].GetParticle(j).SetPosition(curX[j] + 0.5*k2x[j]);
            particleSystem.cubes[i].GetParticle(j).SetForce(Eigen::Vector3d::Zero());
        }

        particleSystem.deltaTime /= 2;
        particleSystem.ComputeCubeForce(particleSystem.cubes[i]);
        particleSystem.deltaTime *= 2;
```




```
        for (int j = 0; j < PartNum; j++)
        {
            k3v.push_back(particleSystem.deltaTime*particleSystem.cubes[i].GetParticle(j).Acceleration());
            k3x.push_back(particleSystem.deltaTime*particleSystem.cubes[i].GetParticle(j).Velocity());
            particleSystem.cubes[i].GetParticle(j).SetVelocity(curV[j] + k3v[j]);
            particleSystem.cubes[i].GetParticle(j).SetPosition(curX[j] + k3x[j]);
            particleSystem.cubes[i].GetParticle(j).SetForce(Eigen::Vector3d::Zero());
        }
        particleSystem.ComputeCubeForce(particleSystem.cubes[i]);
        for (int j = 0; j < PartNum; j++)
        {
            k4v.push_back(particleSystem.deltaTime*particleSystem.cubes[i].GetParticle(j).Acceleration());
            k4x.push_back(particleSystem.deltaTime*particleSystem.cubes[i].GetParticle(j).Velocity());
        }
        for (int j = 0; j < PartNum; j++)
        {
            particleSystem.cubes[i].GetParticle(j).SetVelocity(curV[j]
                + k1v[j].operator/(6.0)
                + k2v[j].operator/(3.0)
                + k3v[j].operator/(3.0)
                + k4v[j].operator/(6.0));
            particleSystem.cubes[i].GetParticle(j).SetPosition(curX[j]
                + k1x[j].operator/(6.0)
                + k2x[j].operator/(3.0)
                + k3x[j].operator/(3.0)
                + k4x[j].operator/(6.0));
        }
    }
}
```

- Discussion:

歐拉積分法與中點法皆為一階近似(取一次導數)，差異僅在取導數的位置不同：

顯性歐拉：起點／中點法：中點／隱性歐拉：終點

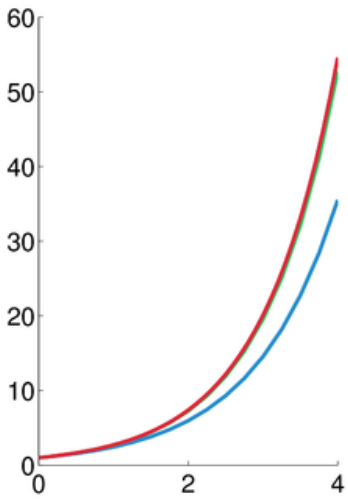
三種方法單步誤差各為

顯性歐拉：Δt^2／中點法：Δt^3／隱性歐拉：Δt^2

右圖為以顯性歐拉(藍)和中點法(綠)近似紅色曲線$\begin{cases} y = e^t \\ y(0) = 1 \end{cases}$示意圖，

x 軸為 t，y 軸為 y(t)值。

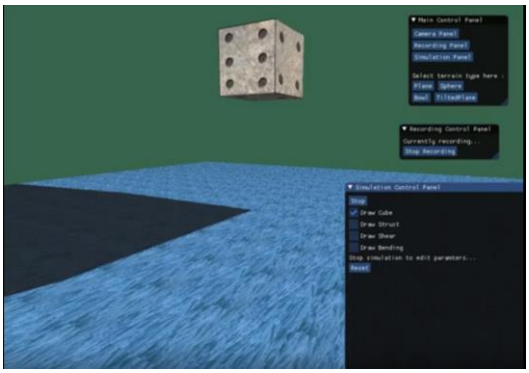
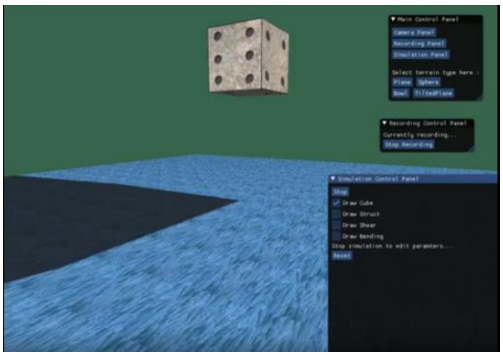
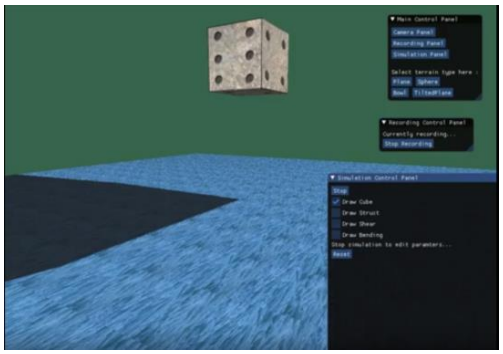
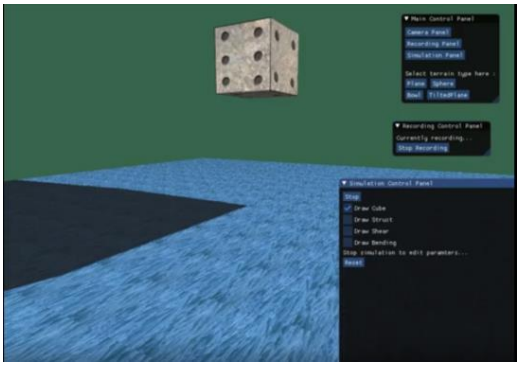
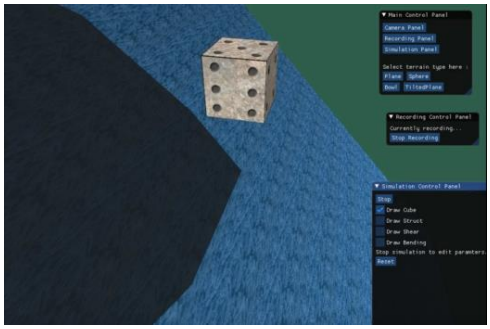
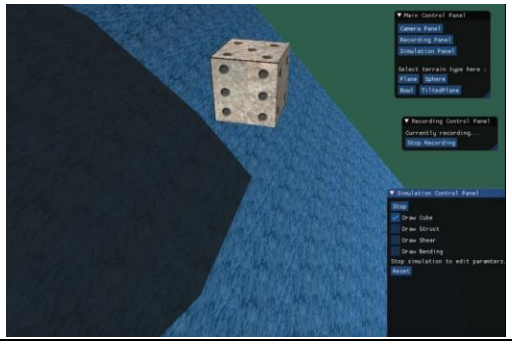
中點積分法相較顯隱性歐拉法具有更快的收斂速度(由右圖$y = e^t$(紅)、中點法(綠/步長$\Delta t = 0.25$)、顯性歐拉(藍/步長$\Delta t = 0.25$)可看出中點積分法收斂較快)。

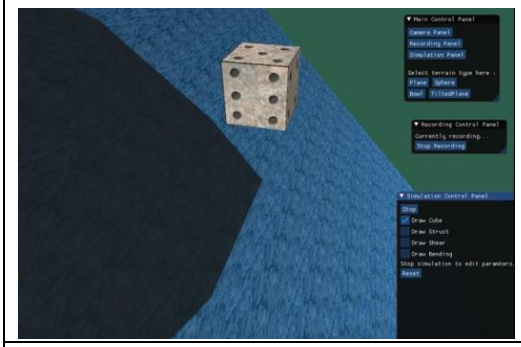
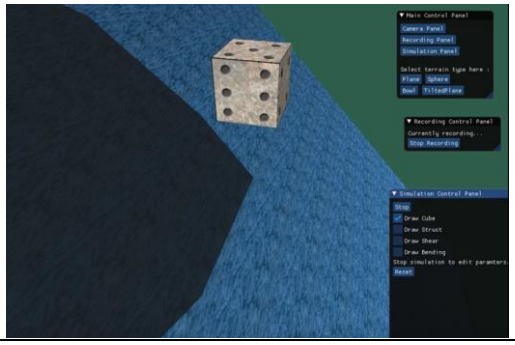
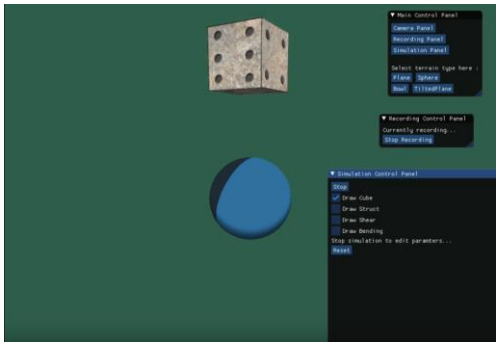
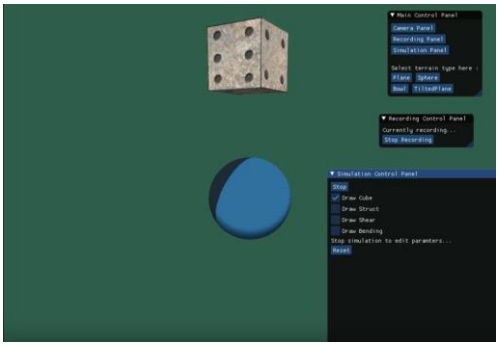
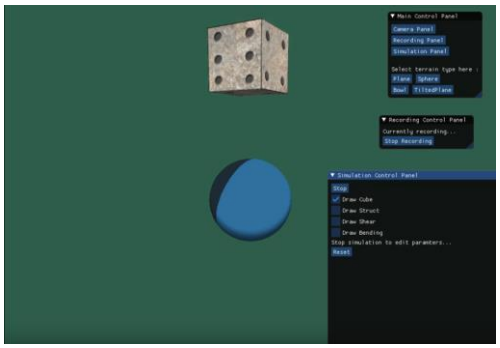
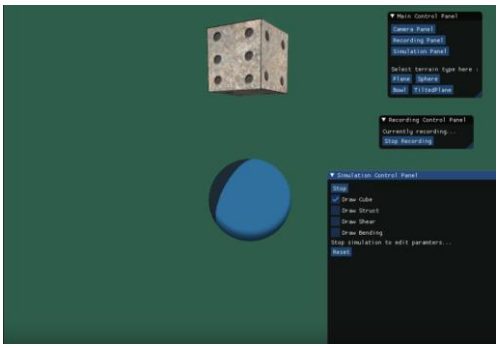
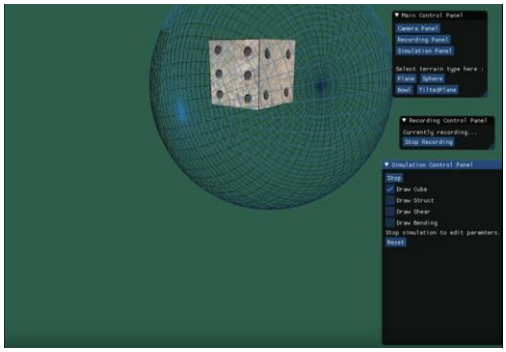
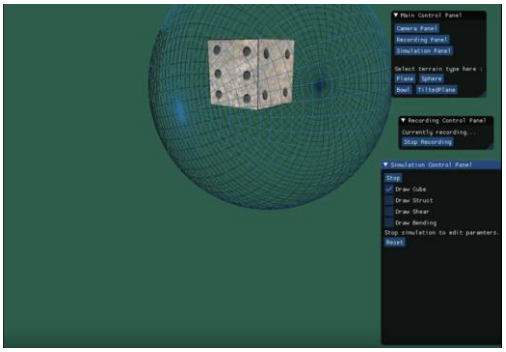


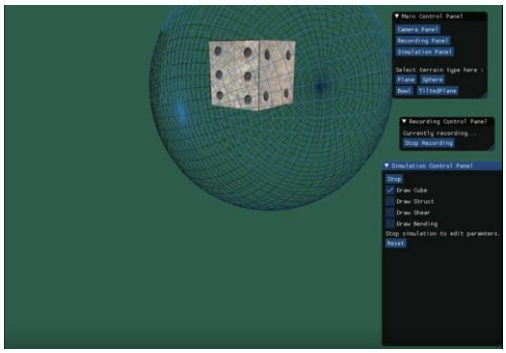
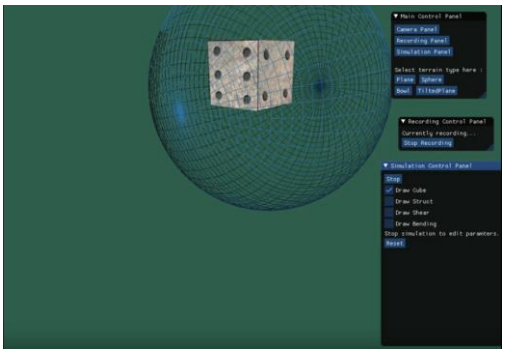
RK4 法是四階方法，每步誤差Δt^5 階，總積累誤差為Δt^4階。

對於 plane terrain，使用顯性歐拉法積分的最大 delta time 為 0.0020 秒／中點法 0.0045 秒／隱性歐拉 0.0020 秒／四階龍格庫塔 0.0030 秒。

Simulation Video:

plane_explicit_euler 	plane_implicit_euler 
plane_midpoint_euler 	plane_RK4 
titled_plane_explicit_euler 	titled_plane_implicit_euler 

titled_plane_ midpoint_euler 	titled_plane_ RK4 
sphere_explicit_euler 	sphere_implicit_euler 
sphere_ midpoint _euler 	sphere_ RK4 
bowl_explicit_euler 	bowl_implicit_euler 

bowl_ midpoint _euler	bowl_ RK4
	

心得:

0710026 薛立甜

這次專題主題是 Soft body simulation, 對於物理不太擅長、也沒有學過微分方程的我來說是個蠻大的挑戰，剛開始看了很久的 power point 都沒看懂，很多地方甚至理解錯誤，還好後來在助教協助下進度才有了顯著的進展，然後又進入了 debug 地獄每天和煒恩在寢室奮鬥，code 刪刪改改，弄了很久還是一直 system unstable，骰子各種奇形怪狀撕裂爆炸我們都見過了 QQ 最後四種 Integrator 搭配四種地形都能穩定模擬的時候超級感動感謝。最後回頭看發現這次專題其實並沒有原本想像中那麼可怕，能順利完成它真是太感謝了。

做完專題覺得更佩服動畫產業工作者了，原來一段動畫光是幾秒鐘就能是他們好幾個小時甚至好幾天好幾個月的心血，以後會更珍惜看的每一部動畫、每一個細節的。能有這種機會貼近動畫產業很開心，謝謝林文杰教授、施亦謙助教、趴呢煒恩。

0710019 黃煒恩

學期初和老薛討論專題主題時，因為「電腦動畫」感覺蠻酷的，所以就找了林文杰教授希望能做與動畫相關的專題。在閱讀電腦動畫模擬相關的資料時，赫然發現隱藏在一個動畫背後的，是許多的模擬公式涉及到物理、數學等等領域，要製作動畫並不簡單。實作完成 Soft body simulation，讓我們了解不同參數以及積分方式模擬對 Soft body 的影響，雖然歷經了些許挫折，但也獲益良多。

完成專題後，我對於電腦動畫的原理有更深入的認識，不再以為動畫只是電視、電腦上觀看的那些有趣的影片。謝謝林文杰教授、施亦謙助教的細心指導，感謝好夥伴老薛的配合和幫助，讓我們得以一起完成這個讓我們既感到苦惱又有趣的專題。