Game Physics in a Nutshell

Massimo Di Pierro

Contents

1 Newton

Everything starts with Newton's Laws which we will phrase as follows:

> Objects like to move at constant velocity. If an object changes its velocity we say it is subject to a force. A force is something external that acts on the object and is proportional to its acceleration. The proportionality factor is a property of the object which we call mass.

Let's measure the position of an object at different times...

time	t	$t+dt$	$t+2dt$	$t+3dt$	$t+4dt$	$t+5dt$	...
position	$\vec{p}_t$	$\vec{p}_{t+dt}$	$\vec{p}_{t+2dt}$	$\vec{p}_{t+3dt}$	$\vec{p}_{t+4dt}$	$\vec{p}_{t+5dt}$	...

We define the velocity as the change in position per unit of time:

$$\vec{v}_t \equiv \frac{\vec{p}_{t+dt} - \vec{p}_t}{dt} \tag{1}$$

and we define the acceleration as the change in velocity per unit of time

$$\vec{a}_t \equiv \frac{\vec{v}_{t+dt} - \vec{v}_t}{dt} \tag{2}$$

time	t	$t+dt$	$t+2dt$	...
position	$\vec{p}_t$	$\vec{p}_{t+dt}$	$\vec{p}_{t+2dt}$	...
velocity	$\vec{v}_t = \frac{\vec{p}_{t+dt}-\vec{p}_t}{dt}$	$\vec{v}_{t+dt} = \frac{\vec{p}_{t+2dt}-\vec{p}_{t+dt}}{dt}$	...	...
acceleration	$\vec{a}_t = \frac{\vec{v}_{t+dt}-\vec{v}_t}{dt}$	...	...	...

If we can determine $\vec{p}$ at different times, we can compute v and a at different times. Notice that if the velocity is constant the acceleration is zero:

$$\vec{a}_t \equiv (\vec{v}_{t+dt} - \vec{v}_t)/dt = 0 \tag{3}$$

We now solve eq. 1 and eq. 2 in $\vec{p}_{t+dt}$ and $\vec{v}_{t+dt}$ respectively. We find:

$$\vec{p}_{t+dt} = \vec{p}_t + \vec{v}_t dt \tag{4}$$

$$\vec{v}_{t+dt} = \vec{v}_t + \vec{a}_t dt \tag{5}$$

The Newton's laws (as stated above) say that if the velocity changes. I.e. there is an acceleration, then the object is subject to a force. The law does not say that if two objects are subject to the same force, they will have the same acceleration. Therefore, it is reasonable to assume that

$$\vec{F}_t \propto \vec{a}_t \tag{6}$$

and the proportionality factor is different for different objects. We call this proportionality factor m:

$$\vec{F}_t = m\vec{a}_t \tag{7}$$

We replace the solution of eq. 7 for a_t in eq. 5 and we obtain:

$$\vec{p}_{t+dt} = \vec{p}_t + \vec{v}_t dt \tag{8}$$

$$\vec{v}_{t+dt} = \vec{v}_t + (1/m)\vec{F}_t dt \tag{9}$$

In other words: *if we know the position and the velocity of the object, and the force acting on the object at time t we can compute the position and the velocity at time $t + dt$.*

The set of eqs. 8 and 9 are called an *Euler integrator*. Technically they are only correct in the limit $dt \to 0$ but they provide a decent approximation for small dt if the force does not change significnaly over the time interval dt.

2 Revised Newton

Is m constant? What if m changes with time? A more accurate way to write the Newton equation is in terms of a quantity we call *momentum*, defined as:

$$\vec{K}_t \equiv m_t \vec{v}_t \tag{10}$$

In terms of $\vec{K}_t$, eq. 7 can be written as

$$\vec{F}_t = \frac{\vec{K}_{t+dt} - \vec{K}_t}{dt} \tag{11}$$

Now we can perform a change of variables from v to K and rewrite the Euler integrator as follows:

$$\vec{F}_t = \sum_i \vec{F}_i \quad \text{(compute force)} \tag{12}$$

$$\vec{v}_t = m_t^{-1}\vec{K}_t \quad \text{(compute velocity)} \tag{13}$$

$$\vec{p}_{t+dt} = \vec{p}_t + \vec{v}_t dt \quad \text{(update position)} \tag{14}$$

$$\vec{K}_{t+dt} = \vec{K}_t + \vec{F}_t dt \quad \text{(update momentum)} \tag{15}$$

In other words: *if we know the position and the momentum of the object, and the force acting on the object at time t, we can compute the position and the momentum at time $t + dt$. If we know the mass of the object we can compute the velocity from the momentum.*

If multiple forces act of the same object, $\vec{F}$ is the sum of those forces:

$$\vec{F} = \sum_i \vec{F}_i \tag{16}$$

where $\vec{F}_i$ is the force caused by interation i (could be gravity, could be a spiring, etc.). This called *d'Alambert's principle*.

3 Types of forces

3.1 Gravity

$$\vec{F}_t^{gravity} \equiv (0, -m_t g, 0) \tag{17}$$

where $g = 9.8$meters/second.

3.2 Spring

$$\vec{F}_t^{spring} \equiv \kappa(L - |\vec{p}_t - \vec{q}_t|)\frac{\vec{p}_t - \vec{q}_q}{|\vec{p}_t - \vec{q}_t|} \tag{18}$$

where $\vec{p}_t$ is the position of the mass at time t, q_t is the position of the other end of the spring at the same time, κ is a constant that describes the rigidity of the spring, and L is the rest length of the pring. Notice that when $|\vec{p}_t - \vec{q}_t| = L$ there is no force, when $|\vec{p}_t - \vec{q}_t| > L$ the force is attractive and when $|\vec{p}_t - \vec{q}_t| < L$ the force is repulsive.

3.3 Friction

$$\vec{F}_t^{friction} \equiv -\gamma \vec{v}_t \tag{19}$$

γ is called the *friction coefficient*.

4 Rotations

A position vector $\vec{p}$ can be multiplied by a matrix

$$\vec{p'} = R\vec{p} \tag{20}$$

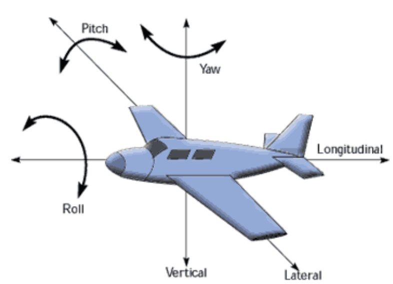
The matrix multiplication maps one vector p onto another vector p'. If we now require that the matrix R have determinant equal to 1 it follows that for every vector p

$$|\vec{p'}| = |R\vec{p}| = |R||\vec{p}| = |\vec{p}| \tag{21}$$

i.e. the R transformation preserves the vector length. If we also requires that the inverse of R be the same as its transposed ($R^{-1} = R^t$) then, for every two vectors p and q we obtain:

$$\vec{p'} \cdot \vec{q'} = (R\vec{p}) \cdot (R\vec{q}) = \vec{p} \cdot \vec{q} \tag{22}$$

i.e. the R transformation preserves scalar products (and therefore angles). A matrix R meeting the two conditions above is called an *orthogonal matrix* or a *rotation*.



A rotation of the angle θ around the X-axes can be written as

$$R_X(\theta) \equiv \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{pmatrix} \tag{23}$$

A rotation of the angle θ around the Y-axes can be written as

$$R_Y(\theta) \equiv \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix} \tag{24}$$

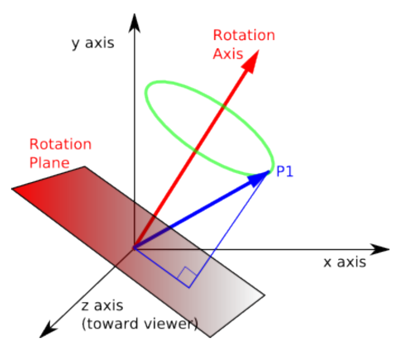
A rotation of the angle θ around the Z-axes can be written as

$$R_Z(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{25}$$

A rotation of the angle θ around an arbitrary direction $\hat{n} = (n_0, n_0, n_1)$ is given by:

$$R_{\hat{n}}(\theta) \equiv \begin{pmatrix} 2(n_0^2 s^2 + c^2) - 1 & 2(n_0 n_1 s^2 - n_2 sc) & 2(n_0 n_2 s^2 - n_1 sc) \\ 2(n_0 n_1 s^2 + n_2 sc) & 2(n_1^2 s^2 + c^2) - 1 & 2(n_1 n_2 s^2 - n_0 sc) \\ 2(n_0 sc - n_1 sc) & 2(n_0 sc + n_1 n_2 s^2) & 2(n_2^2 s^2 + c^2) - 1 \end{pmatrix} \tag{26}$$

where $c = \cos(\theta/2)$ and $s = \sin(\theta/2)$.



For later convenience we will define a four dimension vector (marked by a tilde):

$$\tilde{\theta} = (\theta_0, \theta_1, \theta_2, \theta_3) = (n_0 \sin(\theta/2), n_1 \sin(\theta/2), n_2 \sin(\theta/2), \cos(\theta/2)) \tag{27}$$

and with a change of variable the matrix of rotation $R_{\hat{n}}(\theta)$ can be re-written:

$$R(\tilde{\theta}) \equiv \begin{pmatrix} 2(\theta_0\theta_0 + \theta_1\theta_1) - 1 & 2(\theta_0\theta_1 - \theta_2\theta_3) & 2(\theta_0\theta_2 - \theta_1\theta_3) \\ 2(\theta_0\theta_1 + \theta_2\theta_3) & 2(\theta_1\theta_1 + \theta_3\theta_3) - 1 & 2(\theta_1\theta_2 - \theta_0\theta_3) \\ 2(\theta_0\theta_3 - \theta_1\theta_3) & 2(\theta_0\theta_3 + \theta_1\theta_2) & 2(\theta_2\theta_2 + \theta_3\theta_3) - 1 \end{pmatrix} \tag{28}$$

The 4-vector $\tilde{\theta}$ defines an *orientation* and it is called a *quaternion* although its methematical properties are not relevant here and are not discussed. The associated matrix $R(\tilde{\theta})$ can be used to rotates the points of an object into an orientation.

5 Composition of rotations

If a body is oriented according to $R(\tilde{\theta})$ and it gets rotated by another matrix $R_{\hat{n}}(\beta)$ the body will end up being oriented in a different direction:

$$R(\tilde{\theta}') = R_{\hat{n}}(\beta)R(\tilde{\theta}) \tag{29}$$

What's $\tilde{\theta}'$ as function of $\hat{n}$, β and $\tilde{\theta}$? An explicit calculation (omitted) reveals:

$$\tilde{\theta}' = \tilde{\theta} + \frac{1}{2}\vec{\beta}\tilde{\theta} \tag{30}$$

where $\vec{\beta}\tilde{\theta}$ is defined as

$$\vec{\beta}\tilde{\theta} = \left(\begin{array}{c} -(\beta_0\theta_0 + \beta_1\theta_1 + \beta_2\theta_2) \\ (\beta_0\theta_3 + \beta_1\theta_2 - \beta_2\theta_1) \\ (\beta_1\theta_3 + \beta_2\theta_0 - \beta_0\theta_2) \\ (\beta_2\theta_3 + \beta_0\theta_1 - \beta_1\theta_0) \end{array} \right) \tag{31}$$

where $(\beta_0, \beta_1, \beta_2) = (\beta n_0, \beta n_1, \beta n_2)$

Eq. 30 will come up handly later when talking about spinning objects. In this case for an infinitesimal rotation $\beta = \omega dt$:

$$\tilde{\theta}' = \tilde{\theta} + \frac{1}{2}\vec{\omega}\tilde{\theta}dt \tag{32}$$

6 Spinning

We now consider a rotation propotional to time, t. In particular a rotation around a direction $\hat{n} = (n_0, n_1, n_2)$ of an angle $\theta_t = \omega t$ (θ depends on t). Repeating the previous steps and replacing θ_t for θ in eq. 27 we obtain:

$$\tilde{\theta}_t \equiv (n_0 \sin(\omega t/2), n_1 \sin(\omega t/2), n_2 \sin(\omega t/2), \cos(\omega_t/2)) \tag{33}$$

So if we now consider the position of a vector $\vec{p}$ subject to rotation we obtain:

$$\vec{p}_t = R(\tilde{\theta}_t)\vec{p} \tag{34}$$

and we can compute it velocity using the definition:

$$\begin{aligned} \vec{v}_t &= (\vec{p}_{t+dt} - \vec{p}_t)/dt & (35) \\ &= (R(\tilde{\theta}_{t+dt}t)\vec{p} - R(\tilde{\theta}_t)\vec{p})/dt & (36) \\ &= \vec{\omega} \times \vec{p} & (37) \end{aligned}$$

where

$$\vec{\omega} \equiv (\omega n_0, \omega n_1, \omega n_2) \tag{38}$$

is called angular velocity.

7 Newton Second Law for Rotation

Before we have formulated the second law of Newton as

$$\vec{F}_t = \frac{\vec{K}_{t+dt} - \vec{K}_t}{dt} \tag{39}$$

where $\vec{K}_t = m_t \vec{v}_t$. We now consider a rigid body comprised of one mass m at position $\vec{p}$, connected by a solid rod at the origin of the axes. The mass is subject to a force $\vec{F}$.

If there were no rod, the mass would move according to the Newton equation above. Because of the rod, the mass is not free and it feels only the component of the force ortoghonal to the direction of the rod $\vec{r}$ which is the same as the position of the mass $\vec{p}$ because the rod is pinned at the origin of the axes. In order to apply Newton law only to the components orthogonal to r we perform a cross product of both terms

$$\begin{aligned} \vec{r}_t \times \vec{F}_t &= \vec{r}_t \times \frac{\vec{K}_{t+dt} - \vec{K}_t}{dt} && (40) \\ &= \frac{\vec{r}_t \times \vec{K}_{t+dt} - \vec{r}_t \times \vec{K}_t}{dt} && (41) \end{aligned}$$

and if we define the *torque* as

$$\vec{\tau} \equiv \vec{r} \times \vec{F} \tag{42}$$

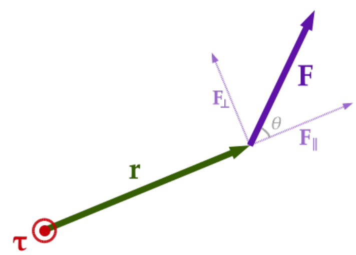
and the *angular momentum* as

$$\vec{L} \equiv \vec{r} \times \vec{K} = m\vec{r} \times \vec{v} \tag{43}$$

we can rewrite eq. 41 as

$$\vec{\tau}_t = \frac{L_{t+dt} - L_t}{dt} \tag{44}$$

which is the anologous of *Newton equation for rotations.*



Notice that if we substiture eq. 37 which gives the velocity in terms of the angualar velocity into eq. 43 we obtain:

$$\vec{L} = \vec{r} \times (m\vec{\omega} \times \vec{r}) = (m|\vec{r}|^2)\vec{\omega} \tag{45}$$

The coefficient $(m|\vec{r}|^2)$ is called *moment of inertia*.

For a rigid body comprised of many masses rotating at the origin of the axes we can define:

$$\vec{\tau} \equiv \sum_i \vec{r}_i \times \vec{F}_i \tag{46}$$

and

$$\vec{L} \equiv \sum_i \vec{r}_i \times \vec{K}_i = m_i \vec{r}_i \times \vec{v}_i \tag{47}$$

and eq. 44 is still valid, while eq. 47 becomes:

$$\vec{L} = \sum_i \vec{r}_i \times (m_i \vec{\omega} \times \vec{r}_i) = I\vec{\omega} \tag{48}$$

where I is a 3x3 matrix (technically a tensor) of components:

$$I_{jk} \equiv \sum_i m_i (\delta_{jk} |\vec{r}_i|^2 - r_{i,j} r_{i,k}) \tag{49}$$

Here $r_{i,j}$ is the j-th component (X=0,Y=1, or Z=2) of vector $\vec{r}_i$.

Finally we can augment the Euler integrator with equivelent formulas for rotations:

$$\vec{F}_t = \sum_i \vec{F}_i \quad \text{(compute force)} \tag{50}$$

$$\vec{\tau}_t = \sum_i \vec{r}_i \times \vec{F}_i \quad \text{(compute torque)} \tag{51}$$

$$\vec{v}_t = m_t^{-1} \vec{K}_t \quad \text{(compute velocity)} \tag{52}$$

$$\vec{\omega}_t = I_t^{-1} L_t \quad \text{(compute angular velocity)} \tag{53}$$

$$\vec{p}_{t+dt} = \vec{p}_t + \vec{v}_t dt \quad \text{(update position)} \tag{54}$$
$$\vec{K}_{t+dt} = \vec{K}_t + \vec{F}_t dt \quad \text{(update momentum)} \tag{55}$$
$$R_t = R(\omega_t dt) R_t \quad \text{(update orientation)} \tag{56}$$
$$\vec{L}_{t+dt} = \vec{L}_t + \vec{\tau}_t dt \quad \text{(update angular momentum)} \tag{57}$$

In other words: *If we know the state of the rigid body characterized by the position p_t and momentum K_t of its center of mass, if we know its orientation T_t and its angular momentum L_t, and if we know the total force F_t and the total torque τ_t acting on the body, we can compute new state of the rigid body (p, K, R, L) at time $t + dt$.*

Given the position p_t of the center of mass of the body and its orientation $\tilde{\theta}_t$, a generic point $\vec{r}_i$ of the body, can be rotated and translated into the global reference frame according to:

$$R_t \vec{r}_i + p_t \tag{58}$$

and the velocity of the point is

$$\vec{\omega}_t \times \vec{r}_i + \vec{v}_t \tag{59}$$

8 Assembling Rigid Bodies

Consider a composite rigid body whose parts are smaller rigid bodies characterized by

Component	Position	Mass	Moment of Inertial
0	$\vec{p}_0$	m_0	I_0
1	$\vec{p}_1$	m_1	I_1
2	$\vec{p}_2$	m_2	I_2
...	...	...	...

The composite object will have:

$$m_{total} = \sum_i m_i \tag{60}$$
$$\vec{p}_{cm} = \sum \vec{p}_i m_i / m_{total} \tag{61}$$
$$I_{total} = \sum_i I_i + \Delta I(m_i, \vec{p}_i - \vec{p}_{total}) \tag{62}$$

where

$$\Delta I(m, \vec{r})_{jk} \equiv m(\delta_{jk} |\vec{r}|^2 - r_j r_k) \tag{63}$$

9 Collisions

Let's first consider particles instead of rigid bodies.

Given the velocities of two bodies $\vec{v}_A$ and $\vec{v}_B$ before a collision we want to determine the velocities of the bodies after the collision, $\vec{v}'_A$ and $\vec{v}'_B$. The separating velocity (relative velocity after the collision) is defined as:

$$\vec{v}_{separating} \equiv \vec{v}'_B - \vec{v}'_A \tag{64}$$

If it is a faction of the closing velocity (relative velocity before the collision):

$$\vec{v}_{separating} = -c\vec{v}_{closing} \tag{65}$$

where c is a restitution coefficient and

$$\vec{v}_{closing} \equiv \vec{v}_B - \vec{v}_A \tag{66}$$

Conservation of momentum and the above relation between closing and separating velocity yields:

$$\vec{v}'_A = \vec{v}_A + d\vec{F}/m_A \tag{67}$$

$$\vec{v}'_B = \vec{v}_B - d\vec{F}/m_B \tag{68}$$

where

$$d\vec{F} = \frac{(c+1)(\vec{v}_B - \vec{v}_A)}{1/m_A + 1/m_B} \tag{69}$$

In the case of collision of the object A with a plane B or other object of infinite mass

$$\vec{v}'_A = \lim_{m_B \longrightarrow \infty} \vec{v}_A + \frac{d\vec{F}}{m_A} = \vec{v}_A + (c+1)(\vec{v}_B - \vec{v}_A) \tag{70}$$

Also notice that if the two bodies get stuck to each other during the collision, then $c = 0$ and

$$\vec{v}'_A = \vec{v}_A + \frac{m_B}{m_A + m_B}(\vec{v}_B - \vec{v}_A) \tag{71}$$

$$\vec{v}'_B = \vec{v}_B + \frac{m_A}{m_A + m_B}(\vec{v}_A - \vec{v}_B) = \vec{v}'_A \tag{72}$$

the two bodies will be moving at the same velocity (the center of mass velocity).

In the case of rigid bodies eq. 65 applies to the points of contact between two rigid bodies. To properly consider its effect on the entire body we have to rewrite the velocity in terms of the velocities of the center of mass plus a component due to the angular velocity.

$$d\vec{F} = \frac{-(c+1)(\vec{v}_{cB} - \vec{v}_{cA}) \cdot \hat{n}}{1/m_A + 1/m_B + |\vec{r}_{cA} \times \hat{n}|^2/I_A + |\vec{r}_{cB} \times \hat{n}|^2/I_B}\hat{n} \tag{73}$$

where $\vec{r}_{cA}$ is the point of collision of body A respect to the center of mass $\vec{p}_A$, $\vec{v}_{cA}$ is the velocity of the collision point before the collision, m_A is the mass of body A, I_A is its moment of inertia. $\hat{n}$ is the direction of $(\vec{p}_B + \vec{r}'_B) - (\vec{p}_A + \vec{r}'_B)$, i.e. the norm ortogonal to the contact. Notice that $\vec{r}_{cA} = R_A\vec{r}_A$ and $\vec{v}_{cA} = \vec{\omega}_A \times \vec{r}_A + \vec{v}_A$. The same for B.

Once this impluse is computed we can deal with the it for each body by adding the contribution of the impulse to force and torque.

$$\begin{aligned}
\vec{F}_A &= ... + d\vec{F}/dt && (74)\\
\vec{\tau}_A &= ... + \vec{r}_{cA} \times d\vec{F}/dt && (75)\\
\vec{F}_B &= ... - d\vec{F}/dt && (76)\\
\vec{\tau}_B &= ... - \vec{r}_{cB} \times d\vec{F}/dt && (77)
\end{aligned}$$

For if two objects we do the following:

- find the point of contact and compute r_{cA}, r_{cB}
- find the compenetration and shift the objects back to cancel compenetration
- compute the impulse (eq. 73)
- update the force and torque to include one time contributon of the impulse (eq. 77).

10 Implementation

Import the necessary libraries

```
#include "fstream"
#include <sstream>
#include <string>
#include <iostream>

#include "math.h"
#include "vector"
#include "list"

#if defined(_MSC_VER)
#include <gl/glut.h>
#else
#include <GLUT/glut.h>
#endif

using namespace std;
```

Define constants

```
#define array vector // to avoid name collions
#define forXYZ(i) for(int i=0; i<3; i++)
const float PRECISION = 0.00001;
const int X=0;
const int Y=1;
const int Z=2;
const int W=3; // for quaternions only
const float gravity = 9.8; // meters/second^2
```

Define class Vector

```
class Vector {
public:
  float v[3];
  Vector(float x=0, float y=0, float z=0) {
    v[X]=x; v[Y]=y; v[Z]=z;
  }
  float operator()(int i) const { return v[i]; }
  float &operator()(int i) { return v[i]; }
};
```

Define operations between vectors

```
Vector operator*(float c, const Vector &v) {
  return Vector(c*v(X),c*v(Y),c*v(Z));
}
Vector operator*(const Vector &v, float c) {
  return c*v;
}
Vector operator/(const Vector &v, float c) {
  return (1.0/c)*v;
}
Vector operator+(const Vector &v, const Vector &w) {
  return Vector(v(X)+w(X),v(Y)+w(Y),v(Z)+w(Z));
}
Vector operator-(const Vector &v, const Vector &w) {
  return Vector(v(X)+w(X),v(Y)+w(Y),v(Z)+w(Z));
```

```
}
float operator*(const Vector &v, const Vector &w) {
  return v(X)*w(X) + v(Y)*w(Y) + v(Z)*w(Z);
}
Vector cross(const Vector &v, const Vector &w) {
  return Vector(v(Y)*w(Z)-v(Z)*w(Y),
         v(Z)*w(X)-v(X)*w(Z),
         v(X)*w(Y)-v(Y)*w(Z));
}
```

Class Rotation

```
class Matrix {
public:
  float m[3][3];
  Matrix() { forXYZ(i) forXYZ(j) m[i][j]=0; }
  const float operator()(int i, int j) const { return m[i][j]; }
  float &operator()(int i, int j) { return m[i][j]; }
};

class Rotation : public Matrix {
public:
  Rotation() { m[X][X]=m[Y][Y]=m[Z][Z]=1; }
  Rotation(const Vector& v) {
    float theta = sqrt(v*v);
    if(theta<PRECISION) {
      forXYZ(i) forXYZ(j) m[i][j] = (i==j)?1:0;
    } else {
      float s = sin(theta), c=cos(theta);
      float t = 1-c;
      float x = v(X)/theta, y = v(Y)/theta, z = v(Z)/theta;
      m[X][X]=t*x*x+c;   m[X][Y]=t*x*y+s*z; m[X][Z]=t*x*z-s*y;
      m[Y][X]=t*x*y-s*z; m[Y][Y]=t*y*y+c;   m[Y][Z]=t*y*z+s*x;
      m[Z][X]=t*x*z+s*y; m[Z][Y]=t*y*z-s*x; m[Z][Z]=t*z*z+x;
    }
  }
};
```

Class Inertia Tensor

```
class InertiaTensor : public Matrix {};
InertiaTensor operator+(const InertiaTensor &a, const InertiaTensor &b) {
  InertiaTensor c=a;
  forXYZ(i) forXYZ(j) c(i,j)+=b(i,j);
  return c;
}
```

Operations between matrices

```
Vector operator*(const Matrix &R, const Vector &v) {
  return Vector(R(X,X)*v(X)+R(X,Y)*v(Y)+R(X,Z)*v(Z),
         R(Y,X)*v(X)+R(Y,Y)*v(Y)+R(Y,Z)*v(Z),
         R(Z,X)*v(X)+R(Z,Y)*v(Y)+R(Z,Z)*v(Z));
}
Matrix operator*(const Matrix &R, const Matrix &S) {
  Matrix T;
  forXYZ(i) forXYZ(j) forXYZ(k) T(i,j)+=R(i,k)*S(k,j);
  return T;
```

```
}
float det(const Matrix &R) {
  return R(X,X)*(R(Z,Z)*R(Y,Y)-R(Z,Y)*R(Y,Z))
    -R(Y,X)*(R(Z,Z)*R(X,Y)-R(Z,Y)*R(X,Z))
    +R(Z,X)*(R(Y,Z)*R(X,Y)-R(Y,Y)*R(X,Z));
}
Matrix operator/(float c, const Matrix &R) {
  Matrix T;
  float d = c/det(R);
  T(X,X)=(R(Z,Z)*R(Y,Y)-R(Z,Y)*R(Y,Z))*d;
  T(X,Y)=(R(Z,Y)*R(X,Z)-R(Z,Z)*R(X,Y))*d;
  T(X,Z)=(R(Y,Z)*R(X,Y)-R(Y,Y)*R(X,Z))*d;
  T(Y,X)=(R(Z,X)*R(Y,Z)-R(Z,Z)*R(Y,X))*d;
  T(Y,Y)=(R(Z,Z)*R(X,X)-R(Z,X)*R(X,Z))*d;
  T(Y,Z)=(R(Y,X)*R(X,Z)-R(Y,Z)*R(X,X))*d;
  T(Z,X)=(R(Z,Y)*R(Y,X)-R(Z,X)*R(Y,Y))*d;
  T(Z,Y)=(R(Z,Y)*R(X,Y)-R(Z,Y)*R(X,X))*d;
  T(Z,Z)=(R(Y,Y)*R(X,X)-R(Y,X)*R(X,Y))*d;
  return T;
}
```

Class Body (describes a rigid body object)

```
class Body {
public:
  // object shape
  array<Vector> r;  // vertices in local coordinates
  array<array<int> > faces;
  // properties of the body ////////////////////////
  bool locked;        // if set true, don't integrate
  float m;            // mass
  InertiaTensor I;    // moments of inertia (3x3 matrix)
  // state of the body /////////////////////////////
  Vector p;           // position of the center of mass
  Vector K;           // momentum
  Matrix R;           // orientation
  Vector L;           // angular momentum
  // auxiliary variables /////////////////////////
  float inv_m;        // 1/m
  Matrix inv_I;       // 1/I
  Vector F;
  Vector tau;
  Vector v;           // velocity
  Vector omega;       // angular velocity
  array<Vector> Rr; // rotated r's.
  array<Vector> vertices; // rotated and shifted r's
  // forces and constraints
  // ...
  Body(float m=1.0, bool locked=false) {
    this->locked = locked;
    this->m = 1.0;
    I(X,X)=I(Y,Y)=I(Z,Z)=m;
    inv_m = 1.0/m;
    inv_I = 1.0/I;
    R = Rotation();
  };
  void clear() { F(X)=F(Y)=F(Z)=tau(X)=tau(Y)=tau(Z)=0; }
```

```
  void update_vertices();
  void integrator(float dt);
  void loadObj(const string & file);
};
```

rotate and shift all vertices from local to universe

```
void Body::update_vertices() {
  Rr.resize(r.size());
  vertices.resize(r.size());
  for(int i=0; i<r.size(); i++) {
     Rr[i] = R*r[i];
     vertices[i]=Rr[i]+p;
  }
}
```

Euler integrator

```
void Body::integrator(float dt) {
  v     = inv_m*K;
  omega = inv_I*L;
  p     = p + v*dt;                // shift
  K     = K + F*dt;                // push
  R     = Rotation(omega*dt)*R;    // rotate
  L     = L + tau*dt;              // spin
  update_vertices();
}
```

Interface for all forces. The constructor can be specific of the force the apply methods adds the contribution to F and tau

```
class Force {
public:
  virtual void apply()=0;
};
```

Gravity

```
class GravityForce : public Force {
public:
  Body *body;
  float g;
  GravityForce(Body *body, float g = gravity) {
    this->body=body; this->g=g;
  }
  void apply() {
    body->F(Y) -= (body->m)*g;
  }
};

class SpringForce : public Force {
public:
  Body *bodyA;
  Body *bodyB;
  int iA, iB;
  float kappa, L;
  SpringForce(Body *bodyA, int iA, Body *bodyB, int iB,
          float kappa, float L) {
```

```
    this->bodyA=bodyA; this->iA=iA;
    this->bodyB=bodyB; this->iB=iB;
  }
  void apply() {
    Vector d = bodyA->vertices[iA]-bodyB->vertices[iB];
    float n = sqrt(d*d);
    Vector F = kappa*(L-n)*d/n;
    bodyA->F = bodyA->F+F;
    bodyB->F = bodyA->F-F;
    bodyA->tau = bodyA->tau + cross(bodyA->Rr[iA],F);
    bodyB->tau = bodyB->tau - cross(bodyB->Rr[iB],F);
  }
};

class AnchoredSpringForce : public Force {
public:
  Body *body;
  int i;
  Vector pin;
  float kappa, L;
  AnchoredSpringForce(Body *body, int i, Vector pin,
              float kappa, float L) {
    this->body=body; this->i=i;
    this->pin = pin;
  }
  void apply() {
    Vector d = body->vertices[i]-pin;
    float n = sqrt(d*d);
    Vector F = kappa*(L-n)*d/n;
    body->F = body->F+F;
    body->tau = body->tau + cross(body->Rr[i],F);
  }
};
```

Friction (ignores the shape of the body, assumes a sphere)

```
class FrictionForce: public Force {
public:
  Body *body;
  float gamma;
    FrictionForce(Body *body, float gamma) {
      this->body=body; this->gamma=gamma;
    }
    void apply() {
      body->F = body->F-gamma*body->v; // ignores shape
    }
};
```

class that stores all bodies, forces and constraints

```
class Universe {
public:
  list<Body*> bodies;
  list<Force*> forces;
  list<Body*>::iterator body;
  list<Force*>::iterator force;
  // evolve universe
  void evolve(float dt) {
```

```
    // clear forces and troques
    for(body=bodies.begin(); body!=bodies.end(); body++)
      (*body)->clear();
    // compute forces and torques
    for(force=forces.begin(); force!=forces.end(); force++)
      (*force)->apply(); // adds to F and tau
    // integrate
    for(body=bodies.begin(); body!=bodies.end(); body++)
      if(!(*body)->locked)
    (*body)->integrator(dt);
    // handle collisions
    for(body=bodies.begin(); body!=bodies.end(); body++)
      if(!(*body)->locked) {
    if((*body)->p(Y)<0) {
      (*body)->p(Y)=0;
      (*body)->K(Y)=-0.9*(*body)->K(Y);
      (*body)->L = 0.9*(*body)->L;
    }
      }
  }
};
```

Auxiliary functions translate moments of Inertia

```
InertiaTensor dI(float m, const Vector &r) {
  InertiaTensor I;
  float r2 = r*r;
  forXYZ(j) forXYZ(k) I(j,k) = m*((j==k)?r2:0-r(j)*r(k));
  return I;
}
```

Auxliary function to include to merge two bodies needs some more work....

```
Body operator+(const Body &a, const Body &b) {
  Body c;
  c.m = (a.m+b.m);
  c.p = (a.m*a.p + b.m+b.p)/c.m;
  c.K = a.K+b.K;
  c.L = a.L+b.L;
  Vector da = a.p-c.p;
  Vector db = b.p-c.p;
  c.I = a.I+dI(a.m,da)+b.I+dI(b.m,db);
  int n = a.r.size();
  // copy all r
  for(int i=0; i<n; i++)
    c.r.push_back(a.r[i]+da);
  for(int i=0; i<b.r.size(); i++)
    c.r.push_back(b.r[i]+db);
  // copy all faces and re-label r
  int m = a.faces.size();
  c.faces.resize(a.faces.size()+b.faces.size());
  for(int j=0; j<a.faces.size(); j++)
    c.faces[j]=a.faces[j];
  for(int j=0; j<b.faces.size(); j++)
    for(int k=0; k<b.faces[j].size(); k++)
      c.faces[j+m].push_back(b.faces[j][k]+n);
  c.update_vertices();
  return c;
```

```
}

void Body::loadObj(const string & file) {
  ifstream input;
  input.open(file.c_str());
  string line;
  if(input.is_open()) {
    while(input.good()) {
      std::getline(input, line);
      if(line.length()>0) {
    string initialVal;
    istringstream instream;
    instream.str(line);
    instream >> initialVal;
    if(initialVal=="v") {
      float x,y,z;
      instream >> x >> y >> z;
      r.push_back(Vector(x,y,z));
    } else if (initialVal=="f") {
      int v1, v2, v3;
      instream >> v1 >> v2 >> v3;
      array<int> triangle;
      triangle.push_back(v1-1);
      triangle.push_back(v2-1);
      triangle.push_back(v3-1);
      faces.push_back(triangle);
    }
      }
    }
    update_vertices();
  }
}
```

GLUT code below Creates a window in which to display the scene.

```
void createWindow(const char* title) {
  int width = 640;
  int height = 480;
  glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
  glutInitWindowSize(width,height);
  glutInitWindowPosition(0,0);
  glutCreateWindow(title);

  glClearColor(0.9f, 0.95f, 1.0f, 1.0f);
  glEnable(GL_DEPTH_TEST);
  glShadeModel(GL_SMOOTH);

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(60.0, (double)width/(double)height, 1.0, 500.0);
  glMatrixMode(GL_MODELVIEW);
}

Universe myuniverse;
```

Called each frame to update the 3D scene. Delegates to the application.

```
void update() {
```

```
  // evolve world
  float timeStep = 0.016f;  // 60fps fixed rate.
  myuniverse.evolve(timeStep);
  glutPostRedisplay();
}
```

Called each frame to display the 3D scene. Delegates to the application.

```
void display() {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();
  gluLookAt(0.0, 3.5, 8.0,  0.0, 3.5, 0.0,  0.0, 1.0, 0.0);
  glColor3f(0,0,0);

  // draw
  for(list<Body*>::iterator body=myuniverse.bodies.begin();
      body!=myuniverse.bodies.end(); body++) {
    glPushMatrix();
    Vector &pos = (*body)->p;
    glTranslatef(pos.v[X], pos.v[Y], pos.v[Z]);
    glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
    for(int i=0; i<(*body)->faces.size(); i++) {
      glBegin(GL_POLYGON);
      for(int j=0; j<(*body)->faces[i].size(); j++) {
    int k = (*body)->faces[i][j];
    glVertex3fv((*body)->vertices[k].v);
      }
      int k = (*body)->faces[i][0];
      glVertex3fv((*body)->vertices[k].v);
      glEnd();
    }
    glPopMatrix();
  }
  // update the displayed content
  glFlush();
  glutSwapBuffers();
}
```

Called when the display window changes size.

```
void reshape(int width, int height) {
  glViewport(0, 0, width, height);
}
```

Called when a mouse button is pressed. Delegates to the application.

```
void mouse(int button, int state, int x, int y) { }
```

Called when a key is pressed.

```
void keyboard(unsigned char key, int x, int y) {
  // Note we omit passing on the x and y: they are rarely needed.
  //Process keyboard
}
```

Called when the mouse is dragged.

```
void motion(int x, int y) { }
```

The main entry point. We pass arguments onto GLUT.

```
int main(int argc, char** argv) {
  // Set up GLUT and the timers
  glutInit(&argc, argv);

  // Create the application and its window
  createWindow("GPNS");
  Body b = Body();
  b.loadObj("assets/sphere.obj");
  b.p = Vector(0,2,0);
  b.L = Vector(0.1,0,0);
  myuniverse.bodies.push_back(&b);
  myuniverse.forces.push_back(new GravityForce(&b,0.01));

  // Set up the appropriate handler functions
  glutReshapeFunc(reshape);
  glutKeyboardFunc(keyboard);
  glutDisplayFunc(display);
  glutIdleFunc(update);
  glutMouseFunc(mouse);
  glutMotionFunc(motion);

  // Run the application
  glutMainLoop();

  // Clean up the application
  return 0;
}
```

11 Appendix

12 Notation

3D Vectors are indicated with a *vector* on top, as in $\vec{p}$. The norm of a vector is indicated with $|\vec{p}|$ or with the same letter as the vector without decoration, p. The direction of a vector $\vec{p}/p$ is indicated with a *hat*, $\hat{p}$ and it is called a *verson*.

4D Vectors (used here to represent quaternions) are marked with a *tilde* superscript, as in $\tilde{\theta}$.

We use the letter t to label time, the label i to label parts of a system (for example components of a rigid body), the labels $j = 0, 1, 2 = X, Y, Z$ and $k = 0, 1, 2 = X, Y, Z$ to indicate components of a vector. Quaternions also have an extra index $j, k = W = 3$.

12.1 Legend

Name	Symbol
position	$\vec{p}$
velocity	$\vec{v} = \dot{p}$
acceleration	$\vec{a}$
momentum	$\vec{K} = m\vec{v}$
mass	m
force	$\vec{F}$
rotation	R
angular velocity	$\vec{\omega}$
angular accelleration	$\vec{\alpha}$
angular momentum	$\vec{L} = I\vec{\omega}$
moment of inertia	I