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

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

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

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

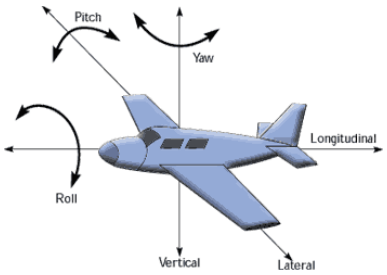
The matrix multiplication maps one vector p onto another vector p'. If we now require that the matrix R have determinant equal to 1 it follows that for every vector p

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

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

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

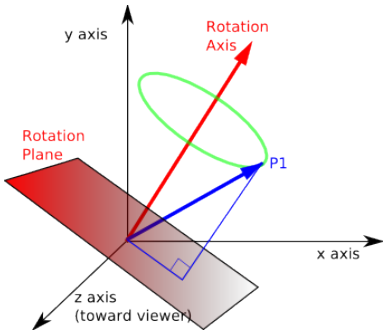
A rotation of the angle θ around the Z-axes can be written as

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

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

$$R_{\hat{n}}(\theta) \equiv \begin{pmatrix} tn_xn_x + c & tn_xn_y - sn_z & tn_xn_z + sn_y \\ tn_xn_y + sn_z & tn_yn_y + c & tn_yn_z - sn_x \\ tn_xn_z - sn_y & tn_yn_z + sn_x & tn_zn_z + c \end{pmatrix} \tag{26}$$

where $c = \cos(\theta)$ and $s = \sin(\theta)$ and $t = (1 - \cos(\theta))$.



5 Composition of rotations

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

$$R_{\hat{n}'}(\theta') = R_{\hat{m}}(\beta)R_{\hat{n}}(\theta) \tag{27}$$

which we can rewrite with with a change of notation as

$$R' = R(\vec{\beta})R \tag{28}$$

where $R = R_{\hat{n}}(\theta)$, $R' = R_{\hat{n}'}(\theta')$, $R(\vec{\beta}) \equiv R_{\hat{m}}(\beta)$ and

$$\vec{\beta} = (\beta m_x, \beta m_y, \beta m_z) \tag{29}$$

For an infinitesimal rotation $\beta = \omega dt$ we can write

$$R' = R(\vec{\omega}dt)R \tag{30}$$

where

$$\vec{\omega} \equiv (\omega n_x, \omega n_y, \omega n_z) \tag{31}$$

is called angular velocity.

6 Spinning

We now consider a rotation propotional to time, t. In particular a rotation around a direction $\hat{n} = (n_x, n_y, n_z)$ of an angle $\theta_t = \omega t$ (θ depends on t). Repeating the previous steps and replacing θ_t for θ in eq. 58 we obtain:

$$\vec{\theta_t} \equiv (\omega t n_x, \omega t n_y, \omega t n_z) \tag{32}$$

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

$$\vec{p_t} = R(\vec{\theta_t})\vec{p} \tag{33}$$

and we can compute it velocity using the definition:

$$\begin{aligned} \vec{v_t} &= (\vec{p}_{t+dt} - \vec{p_t})/dt & (34)\\ &= (R(\vec{\theta}_{t+dt}t)\vec{p} - R(\vec{\theta_t})\vec{p})/dt & (35)\\ &= \vec{\omega} \times \vec{p} & (36) \end{aligned}$$

where $\vec{\omega}$ is the angular velocity, same as eq. 31.

7 Newton Second Law for Rotation

Before we have formulated the second law of Newton as

$$\vec{F_t} = \frac{\vec{K}_{t+dt} - \vec{K}_t}{dt} \tag{37}$$

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

$$\vec{r}_t \times \vec{F}_t = \vec{r}_t \times \frac{\vec{K}_{t+dt} - \vec{K}_t}{dt} \tag{38}$$

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

and if we define the *torque* as

$$\vec{\tau} \equiv \vec{r} \times \vec{F} \tag{40}$$

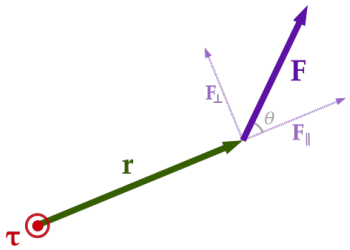
and the *angular momentum* as

$$\vec{L} \equiv \vec{r} \times \vec{K} = m\vec{r} \times \vec{v} \tag{41}$$

we can rewrite eq. 39 as

$$\vec{\tau}_t = \frac{L_{t+dt} - L_t}{dt} \tag{42}$$

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



Notice that if we substiture eq. 36 which gives the velocity in terms of the angualar velocity into eq. 41 we obtain:

$$\vec{L} = \vec{r} \times (m\vec{\omega} \times \vec{r}) = (m|\vec{r}|^2)\vec{\omega} \tag{43}$$

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:

$$\vec{\tau} \equiv \sum_i \vec{r_i} \times \vec{F_i} \tag{44}$$

and

$$\vec{L} \equiv \sum_i \vec{r_i} \times \vec{K_i} = m_i \vec{r_i} \times \vec{v_i} \tag{45}$$

and eq. 42 is still valid, while eq. 45 becomes:

$$\vec{L} = \sum_i \vec{r_i} \times (m_i \vec{\omega} \times \vec{r_i}) = I\vec{\omega} \tag{46}$$

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

$$I_{jk} \equiv \sum_i m_i(\delta_{jk}|\vec{r_i}|^2 - r_{i,j}r_{i,k}) \tag{47}$$

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:

$$\vec{F_t} = \sum_i \vec{F_i} \quad \text{(compute force)} \tag{48}$$

$$\vec{\tau_t} = \sum_i \vec{r_i} \times \vec{F_i} \quad \text{(compute torque)} \tag{49}$$

$$\vec{v_t} = m_t^{-1}\vec{K_t} \quad \text{(compute velocity)} \tag{50}$$

$$\vec{\omega_t} = I_t^{-1}L_t \quad \text{(compute angular velocity)} \tag{51}$$

$$\vec{p}_{t+dt} = \vec{p_t} + \vec{v_t}dt \quad \text{(update position)} \tag{52}$$

$$\vec{K}_{t+dt} = \vec{K_t} + \vec{F_t}dt \quad \text{(update momentum)} \tag{53}$$

$$R_t = R(\omega_t dt)R_t \quad \text{(update orientation)} \tag{54}$$

$$\vec{L}_{t+dt} = \vec{L_t} + \vec{\tau_t}dt \quad \text{(update angular momentum)} \tag{55}$$

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:

$$R_t\vec{r_i} + p_t \tag{56}$$

and the velocity of the point is

$$\vec{\omega}_t \times \vec{r}_i + \vec{v}_t \tag{57}$$

8 Quaternions (optional)

It can be convenient to define a four dimension vector (marked by a tilde):

$$\tilde{\theta} = (\theta_0, \theta_1, \theta_2, \theta_3) = (n_x \sin(\theta/2), n_y \sin(\theta/2), n_z \sin(\theta/2), \cos(\theta/2)) \tag{58}$$

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

$$R(\tilde{\theta}) \equiv \begin{pmatrix} 2(\theta_0\theta_0 + \theta_1\theta_1) - 1 & 2(\theta_0\theta_1 - \theta_2\theta_3) & 2(\theta_0\theta_2 - \theta_1\theta_3) \\ 2(\theta_0\theta_1 + \theta_2\theta_3) & 2(\theta_1\theta_1 + \theta_3\theta_3) - 1 & 2(\theta_1\theta_2 - \theta_0\theta_3) \\ 2(\theta_0\theta_3 - \theta_1\theta_3) & 2(\theta_0\theta_3 + \theta_1\theta_2) & 2(\theta_2\theta_2 + \theta_3\theta_3) - 1 \end{pmatrix} \tag{59}$$

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.

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:

$$R(\tilde{\theta}') = R_{\hat{n}}(\beta) R(\tilde{\theta}) \tag{60}$$

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

$$\tilde{\theta}' = \tilde{\theta} + \frac{1}{2}\vec{\beta}\tilde{\theta} \tag{61}$$

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

$$\vec{\beta}\tilde{\theta} = \begin{pmatrix} -(\beta_0\theta_0 + \beta_1\theta_1 + \beta_2\theta_2) \\ (\beta_0\theta_3 + \beta_1\theta_2 - \beta_2\theta_1) \\ (\beta_1\theta_3 + \beta_2\theta_0 - \beta_0\theta_2) \\ (\beta_2\theta_3 + \beta_0\theta_1 - \beta_1\theta_0) \end{pmatrix} \tag{62}$$

where $(\beta_0, \beta_1, \beta_2) = (\beta n_x, \beta n_y, \beta n_z)$

Eq. 61 will come up handly later when talking about spinning objects. In this case for an infinitesimal rotation $\beta = \omega dt$:

$$\tilde{\theta}' = \tilde{\theta} + \frac{1}{2}\vec{\omega}\tilde{\theta}dt \tag{63}$$

9 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:

$$m_{total} = \sum_i m_i \tag{64}$$

$$\vec{p}_{cm} = \sum \vec{p}_i m_i / m_{total} \tag{65}$$

$$I_{total} = \sum_i I_i + \Delta I(m_i, \vec{p}_i - \vec{p}_{total}) \tag{66}$$

where

$$\Delta I(m, \vec{r})_{jk} \equiv m(\delta_{jk}|\vec{r}|^2 - r_j r_k) \tag{67}$$

10 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:

$$\vec{v}_{separating} \equiv \vec{v}'_B - \vec{v}'_A \tag{68}$$

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

$$\vec{v}_{separating} = -c\vec{v}_{closing} \tag{69}$$

where c is a restitution coefficient and

$$\vec{v}_{closing} \equiv \vec{v}_B - \vec{v}_A \tag{70}$$

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

$$\vec{K}'_A = \vec{K}_A + \vec{J}/m_A \quad (71)$$

$$\vec{K}'_B = \vec{K}_B - \vec{J}/m_B \quad (72)$$

where

$$\vec{J} = \frac{(c+1)(\vec{v}_B - \vec{v}_A)}{1/m_A + 1/m_B} \quad (73)$$

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

$$\vec{K}'_A = \lim_{m_A \longrightarrow \infty} \vec{K}_A + \frac{\vec{J}}{m_A} = \vec{K}_A + m_A(c+1)(\vec{v}_B - \vec{v}_A) \quad (74)$$

In the case of rigid bodies eq. 69 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.

$$\vec{J}_\perp = \frac{-(c+1)(\vec{v}_{cB} - \vec{v}_{cA}) \cdot \hat{n}}{1/m_A + 1/m_B + [(1/I_A)(\vec{r}_{cA} \times \hat{n}) \times r_{cA} + (1/I_B)(\vec{r}_{cB} \times \hat{n}) \times r_{rB}] \cdot \hat{n}} \hat{n} \quad (75)$$

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 a versor orthogonal to the contact point. 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.

In presence of friction at the contact point, there is also a tangential inpulse to take into account:

$$\vec{J}_\parallel = \frac{-(c_f+1)(\vec{v}_{cB} - \vec{v}_{cA}) \cdot \hat{t}}{1/m_A + 1/m_B + \left[(1/I_A)(\vec{r}_{cA} \times \hat{t}) \times r_{cA} + (1/I_B)(\vec{r}_{cB} \times \hat{t}) \times r_{rB}\right] \cdot \hat{t}} \hat{t} \quad (76)$$

where $\hat{t}$ is a versor orthogonal to n and to the relative velocity beween the colliding points. In general c_f is different from c.

$$\vec{K}'_A = K_A + (\vec{J}_\perp + \vec{J}_\parallel) \quad (77)$$
$$\vec{\tau}'_A = \tau_A + \vec{r}_{cA} \times (\vec{J}_\perp + \vec{J}_\parallel) \quad (78)$$
$$\vec{K}'_B = K_B - (\vec{J}_\perp + \vec{J}_\parallel) \quad (79)$$
$$\vec{\tau}'_B = \tau_B - \vec{r}_{cB} \times (\vec{J}_\perp + \vec{J}_\parallel) \quad (80)$$

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. 76)
- update the force and torque to include one time contributon of the impulse (eq. 80).

11 Implementation

Import the necessary libraries:

```
#include "fstream"
#include <sstream>
#include <string>
#include <iostream>
#include "math.h"
#include "vector"
#include "set"
#if defined(_MSC_VER)
#include <gl/glut.h>
#else
#include <GLUT/glut.h>
#endif
using namespace std;
```

Define useful macros and constants:

```
#define array vector // to avoid name collions
#define forXYZ(i) for(int i=0; i<3; i++)
#define forEach(i,s) for(i=(s).begin();i!=(s).end();i++)
#define OBJ(iterator) (*(*iterator))
const float PRECISION = 0.00001;
const int X = 0;
const int Y = 1;
const int Z = 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);
```

```
}
float norm(const Vector &v) {
  return sqrt(v*v);
}
Vector versor(const Vector &v) {
  float d = norm(v);
  return (d>0)?(v/d):v;
}
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(X));
}
```

Class matrix is a base for rotation and inertiatensor:

```
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]; }
};
```

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 rotation is a matrix:

```
class Rotation : public Matrix {
public:
  Rotation() { forXYZ(i) forXYZ(j) m[i][j] = (i==j)?1:0; }
  Rotation(const Vector& v) {
    float theta = norm(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+c;
    }
  }
};
```

Class inertiatensor is also a matrix:

```
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;
}
```

Class body (describes a rigid body object):

```
class Body {
public:
  // object shape
  float radius;      // all vertices inside radius;
  array<Vector> r;   // vertices in local coordinates
  array<array<int> > faces;
  Vector color;      // color of the object;
  // 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) {
    radius = 0;
```

```
    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, float scale);
  void draw();
};
```

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(float dt)=0;
  virtual void draw() {};
};
```

Gravity is a force:

```
class GravityForce : public Force {
public:
  Body *body;
  float g;
  GravityForce(Body *body, float g = 0.01) {
    this->body = body; this->g = g;
  }
  void apply(float dt) {
    body->F(Y) -= (body->m)*g;
  }
};
```

Spring is a force:

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

A spring can be anchored to a pin:

```
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;
    this->kappa = kappa; this->L = L;
  }
  void apply(float dt) {
    Vector d = body->vertices[i]-pin;
    float n = norm(d);
    Vector F = kappa*(n-L)*d/n;
    body->F = body->F+F;
    body->tau = body->tau + cross(body->Rr[i],F);
  }
};
```

Friction is alos a force (this ignores the shape of the body, assumes a sphere):

```
class FrictionForce: public Force {
public:
  Body *body;
  float gamma;
    FrictionForce(Body *body, float gamma) {
```

```
6       this->body = body; this->gamma = gamma;
7     }
8     void apply(float dt) {
9       body->F = body->F-gamma*(body->v)*dt; // ignores shape
10    }
11 };
```

Class water, one instance floods the universe:

```
1  class Water: public Force {
2  public:
3    float level, wave, speed;
4    float m[41][41];
5    float t, x0,dx;
6    set<Body*> *bodies;
7    Water(set<Body*> *bodies, float level,
8      float wave=0.2, float speed=0.2) {
9      this->bodies = bodies;
10     this->level = level; this->wave = wave, this->speed = speed;
11     t = 0; x0 = 5.0; dx = 0.25;
12   }
13   void apply(float dt) {
14     set<Body*>::iterator ibody;
15     t = t+dt;
16     for(int i=0; i<41; i++)
17       for(int j=0; j<41; j++)
18     this->m[i][j] = level+wave/2*sin(speed*t+i)+wave/2*sin(0.5*i*j);
19     forEach(ibody,*bodies) {
20       Body &body = OBJ(ibody); // dereference
21       int i = (body.p(X)+x0)/dx;
22       int j = (body.p(Z)+x0)/dx;
23       if(body.p(Y)<m[i][j]) {
24     body.F = body.F + (Vector(0,1,0)-2.0*(body.v))*dt;
25     body.L = (1.0-dt)*body.L;
26       }
27     }
28   }
29   void draw();
30 };
```

A constraint has detect and resolve methods:

```
1  class Constraint {
2  public:
3    virtual bool detect()=0;
4    virtual void resolve(float dt)=0;
5    virtual void draw() {}
6    Vector impluse(const Body &A, const Body &B,
7           Vector &r_A, Vector &r_B, Vector n, float c) {
8      float IA; // FIX
9      float IB; // FIX
10     Vector r_cA = A.R*r_A;
11     Vector r_cB = B.R*r_B;
12     Vector v_cA = cross(A.omega,r_A)+A.v;
13     Vector v_cB = cross(B.omega,r_B)+B.v;
14     Vector crossA = cross(r_cA,n);
15     Vector crossB = cross(r_cB,n);
16     Vector dF = (-(c-1)/(A.inv_m+B.inv_m+
```

```
            crossA*crossA/IA+
            crossB*crossB/IB)*(v_cB-v_cB)*n)*n;
  }
};
```

Class to deal with collision with static plane (ignore rotation):

```
class PlaneConstraint: public Constraint {
public:
  Body *body;
  Vector n,d;
  float penetration, restitution;
  // d is the discance of the plane from origin
  // n is a versor orthogonal to plane
  // (in opposite direction from collision)
  PlaneConstraint(Body *body, float restitution,
          const Vector &d, const Vector &n) {
    this->body = body;
    this->n = n; this->d = d;
    this->restitution = restitution;
  }
  bool detect() {
    penetration = body->p*n-d*n + body->radius;
    return penetration>=0;
  }
  void resolve(float dt) {
    // move the object back is stuck on plane
    body->p = body->p - penetration*n;
    float K_ortho = n*body->K;
    // optional, deal with friction
    Vector L_ortho = -(body->radius)*cross(n,body->K-K_ortho);
    body->L = (n*body->L)*n + L_ortho;
    // reverse momentum
    if(K_ortho>0)
      body->K = body->K - (restitution+1)*(K_ortho)*n;
  }
};
```

Default all-2-all collision handler:

```
class All2AllCollisions : public Constraint {
public:
  float restitution;
  set<Body*> *bodies;
  All2AllCollisions(set<Body*> *bodies, float c=0.5) {
    this->bodies=bodies;
    restitution = c;
  }
  bool detect() { return true; }
  void All2AllCollisions::resolve(float dt) {
    set<Body*>::iterator ibodyA, ibodyB;
    forEach(ibodyA,*bodies) {
      forEach(ibodyB,*bodies) {
    Body &A = OBJ(ibodyA); // dereference
    Body &B = OBJ(ibodyB); // dereference
    Vector d = B.p-A.p;
    Vector v_closing = B.v-A.v;
    float penetration = A.radius+B.radius-norm(d);
```

```
    if(penetration>0 && v_closing*d<0) {
      Vector q = (penetration/(A.m+B.m))*versor(d);
      A.p = A.p-B.m*q;
      B.p = B.p+A.m*q;
      Vector impulse = (-restitution*A.m*B.m/(A.m+B.m))*v_closing;
      A.K = A.K-impulse;
      B.K = B.K+impulse;
    }
      }
    }
  }
};
```

A universe stores bodies, forces, constraints and evolves in time:

```
class Universe {
public:
  float dt;
  // universe state
  set<Body*> bodies;
  set<Force*> forces;
  set<Constraint*> constraints;
  // useful iterators
  set<Body*>::iterator ibody;
  set<Force*>::iterator iforce;
  set<Constraint*>::iterator iconstraint;
  int frame;
  Universe() {
    frame = 0;
  }
  ~Universe() {
    forEach(ibody,bodies) delete (*ibody);
    forEach(iforce,forces) delete (*iforce);
    forEach(iconstraint,constraints) delete (*iconstraint);
  }
  // evolve universe
  void evolve() {
    // clear forces and troques
    forEach(ibody,bodies)
      OBJ(ibody).clear();
    // compute forces and torques
    forEach(iforce,forces)
      OBJ(iforce).apply(dt);
    callback();
    // integrate
    forEach(ibody,bodies)
      if(!OBJ(ibody).locked)
    OBJ(ibody).integrator(dt);
    // handle collisions (not quite right yet)
    forEach(iconstraint,constraints)
      if(OBJ(iconstraint).detect())
    OBJ(iconstraint).resolve(dt);
    frame++;
  }
public:
  virtual void build_universe()=0;
  virtual void callback()=0;
};
```

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.p-c.p)*a.K+(b.p-c.p)*b.K;
  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;
}
```

Function that loads an wavefront obj file into a body:

```
void Body::loadObj(const string & file,float scale=0.5) {
  ifstream input;
  string line;
  float x,y,z;
  input.open(file.c_str());
  if(input.is_open()) {
    while(input.good()) {
      getline(input, line);
      if(line.length()>0) {
    string initialVal;
    istringstream instream;
    instream.str(line);
    instream >> initialVal;
    if(initialVal=="v") {
      instream >> x >> y >> z;
      Vector p = scale*Vector(x,y,z);
      r.push_back(p);
      radius = max(radius,norm(p));
```

```
    } else if (initialVal=="f") {
      array<int> path;
      while(instream >> x) path.push_back(x-1);
      faces.push_back(path);
    }
      }
    }
    update_vertices();
  }
}
```

Make my universe!:

```
class MyUniverse : public Universe {
public:
  void build_universe() {
    Body *b_old = 0;
    for(int i=0; i<4; i++) {
      Body &b = *new Body();
      b.color=Vector(((i+1)%4)?1:0,i%2,(i%3)?1:0);
      b.loadObj("assets/sphere.obj");
      b.p = Vector(i,i+2,-i);
      b.K = Vector(0.1*i,0.01*i,0);
      b.L = Vector(0.5,0.5*i,0.1*i);
      bodies.insert(&b);
      forces.insert(new GravityForce(&b,0.01));
      constraints.insert(new PlaneConstraint(&b,0.9,Vector(0,0,0),
                         Vector(0,-1,0)));
      constraints.insert(new PlaneConstraint(&b,0.9,Vector(5,0,0),
                         Vector(1,0,0)));
      constraints.insert(new PlaneConstraint(&b,0.9,Vector(-5,0,0),
                         Vector(-1,0,0)));
      constraints.insert(new PlaneConstraint(&b,0.9,Vector(0,0,5),
                         Vector(0,0,1)));
      constraints.insert(new PlaneConstraint(&b,0.9,Vector(0,0,-5),
                         Vector(0,0,-1)));
      // forces.insert(new FrictionForce(&b,0.5));
      if(i==2)
    forces.insert(new SpringForce(b_old,0,&b,0,0.01,0));
      b_old = &b;
    }
    constraints.insert(new All2AllCollisions(&bodies,0.8));
    // forces.insert(new Water(&bodies,3.0));
  }
  void callback() {}
};

MyUniverse universe;
```

Glut code below creates a window in which to display the scene:

```
void createWindow(const char* title) {
  int width = 640, 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);
}
```

Called each frame to update the 3d scene. delegates to the application:

```
void update() {
  // evolve world
  universe.dt = 0.016f; // 60fps fixed rate.
  universe.evolve();
  glutPostRedisplay();
}
```

Function called each frame to display the 3d scene. it draws all bodies, forces and constraints:

```
void display() {
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity();
  gluLookAt(0.0,3.5,10.0, 0.0,3.5,0.0, 0.0,1.0,0.0);
  forEach(universe.ibody,universe.bodies)
    OBJ(universe.ibody).draw();
  forEach(universe.iforce,universe.forces)
    OBJ(universe.iforce).draw();
  forEach(universe.iconstraint,universe.constraints)
    OBJ(universe.iconstraint).draw();
  // update the displayed content
  glFlush();
  glutSwapBuffers();
}
```

Code that draws an body:

```
void Body::draw() {
  glPolygonMode(GL_FRONT,GL_FILL);
  for(int i=0; i<faces.size(); i++) {
    float k = 0.5*(1.0+(float)(i+1)/faces.size());
    glColor3f(color(0)*k,color(1)*k,color(2)*k);
    glBegin(GL_POLYGON);
    for(int j=0; j<faces[i].size(); j++)
      glVertex3fv(vertices[faces[i][j]].v);
    glVertex3fv(vertices[faces[i][0]].v);
    glEnd();
  }
}
```

Code that draws a spring:

```
void SpringForce::draw() {
  glColor3f(0,0,0);
  glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
  glBegin(GL_LINES);
  glVertex3fv(bodyA->vertices[iA].v);
  glVertex3fv(bodyB->vertices[iB].v);
```

```
  glEnd();
}
```

Draw the the water:

```
void Water::draw() {
  glPolygonMode(GL_FRONT,GL_FILL);
  for(int i=0; i<40; i++) {
    glBegin(GL_POLYGON);
    for(int j=0; j<40; j++) {
      glColor3f(0,0,0.5+0.25*(m[i][j]-level+wave)/wave);
      glVertex3fv(Vector(-x0+dx*i,m[i][j],-x0+dx*j).v);
      glVertex3fv(Vector(-x0+dx*i,m[i][j+1],-x0+dx*j+dx).v);
      glVertex3fv(Vector(-x0+dx*i+dx,m[i+1][j+1],-x0+dx*j+dx).v);
      glVertex3fv(Vector(-x0+dx*i+dx,m[i+1][j],-x0+dx*j).v);
    }
    glEnd();
  }
}
```

Function called when the display window changes size:

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

Function called when a mouse button is pressed:

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

Function called when a key is pressed:

```
void keyboard(unsigned char key, int x, int y) {
  // '1' kick ball 1, '2' kicks ball 2, etc.
  int i = 0;
  set<Body*>::iterator ibody;
  forEach(ibody,universe.bodies) {
    if(key-49==i) {
      Body &body = OBJ(ibody); // dereference
      body.K = Vector(0.2,0.2,0);
      body.L = Vector(0,0,0.04);
    }
    i++;
  }
}
```

Called when the mouse is dragged:

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

The main function. everythign starts here:

```
int main(int argc, char** argv) {
  // Create the application and its window
  glutInit(&argc, argv);
  createWindow("Cylon");
  // fill universe with stuff
  universe.build_universe();
  // Set up the appropriate handler functions
```

```
 8   glutReshapeFunc(reshape);
 9   glutKeyboardFunc(keyboard);
10   glutDisplayFunc(display);
11   glutIdleFunc(update);
12   glutMouseFunc(mouse);
13   glutMotionFunc(motion);
14   // Enter into a loop
15   glutMainLoop();
16   return 0;
17 }
```

12 Appendix

13 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 $j, k = W = 3$.

13.1 Legend

Name	Symbol
position	$\vec{p}$
velocity	$\vec{v}$
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