Game Physics in a Nutshell

Massimo Di Pierro

Contents

Source: `https://github.com/mdipierro/cylon`

1 Definitions

Disclaimer : *This document is a work in progress. Here only talk about classical physics (200 years old stuff) and all definitions and formulas are consistent within that framework. Classical physics works well for objects that have relative velocity much smaller than the speed of light and are macroscopic thus not affected by the measurement process.*

The word **Physics** comes the greek *physiké* which means *science of nature*. It is the study of natural phenomena by means of quantitative observations and formal languages such as mathematics. The ultimate goal of physics is that describing the entire universe using few fundamental laws expressed in the mathematical languages.

A **Cartesian coordinate system** is a framework that allows to specify each point uniquely using numerical coordinates, which are the distances from the point to D fixed perpendicular directed lines, measured in the same unit of length. Each reference line is called a coordinate axis or just axis of the system, and the point where they meet is its origin. The coordinates can also be defined as the positions of the perpendicular projections of the point onto the two axes, expressed as signed distances from the origin. Given two points identified by coordinates $\vec{p} = (p_x, p_y, p_z)$ and $\vec{q} = (q_x, q_y, q_z)$, the distance between the two points is given by

$$\sqrt{(p_x - q_x)^2 + (p_y - q_y)^2 + (p_z - q_z)^2} \tag{1}$$

and indicated with $d_{pq} = |\vec{p} - \vec{q}|$.

A **reference frame** or **observational frame of reference** is a set of axes and their orientation, tied together with information about the motion of the observer. For example you can have two observers at the same location at the same moment in time but in motion one respect to the other. They may use the same set of axes and the same orientation yet, when measuring speed of objects respect to their own speed they will find different values. In particular if we get in the shoes of observer O and we measure the speed of point P we find velotiy $\vec{v}$. If we instead get of the shoes of observer O' and we measure the speed of point P we find a velocity $\vec{v}'$. This means that the velocity of observer O' as measured by O is $\vec{w} = \vec{v} - \vec{v}'$ and the velotiy of O

as measured by O' is $\vec{w}' = -\vec{w} = \vec{v}' - \vec{v}$. This law of composition of velocities is called **Galilean Invariance**.

A reference frame is called an **intertial reference frame** if it describes time and space homogeneously, isotropically, and in a time-independent manner. An observer is in an inertial reference frame if the observer feels no force acting upon him/her. All inertial frames are in a state of constant, rectilinear motion with respect to one another. If an observer A in an inertial reference frame and he/she measures the position and velocity of a point P as $\vec{p}, \vec{v}_p$, then a different observer B from a reference frame moving at velocity $\vec{w}$ respect to A will measure different position and velocity for the same point P, $\vec{p}' = \vec{p} - \vec{w}t, \vec{v}'_p = \vec{p} - \vec{w}$. Laws of physics like $F = ma$ are normally formulated in an inertial reference frame and require "corrections" when used in a non-inertial reference frame.

Degrees of freedom are the number of parameters required to specify the state of a system. A particle is only identified by its coordinates (3 numbers in 3D). A rigid body is identified by its position and its orientation (3+3=6 numbers in 3D). A system comprised of n point particles and m constraints has $3n - m$ degrees of freedom. A system comprised of n rigid bodies and m constraints has $6n - m$ degrees of freedom.

2 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 *versor*.

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 $j, k = W = 3$. 4D Vectors (used here to represent quaternions) are marked with a *tilde* superscript, as in $\tilde{\theta}$.

Here is a legend for the notation used in these notes:

Name	Symbol
point	A, B, P, ...
position	$\vec{p}$, $\vec{p}_A$, $\vec{q}$, ...
component of a vector	p_k for $k = 1, 2, 3$ or $k = X, Y, Z$
velocity	$\vec{v}$
acceleration	$\vec{a}$
momentum	$\vec{K} = m\vec{v}$
mass	m
force	$\vec{F}$
rotation	R
angular velocity	$\vec{\omega}$
angular acceleration	$\vec{\alpha}$
angular momentum	$\vec{L} = I\vec{\omega}$
inertia tensor	I

We use the symbol = to indicate en equivalence (result of an observation, consequence of a formula, or an assumption) but we use the symbol $\equiv$ to indicate a definition (the left hand term defined in terms of the right hand term).

3 Newton First and Second Laws

Classical Mechanis starts with Newton's Laws. The first two of them can be summarized as follows:

> When observed from an inertial reference frame, 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:

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

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

$$\vec{a}_t \equiv \frac{\vec{v}_{t+dt} - \vec{v}_t}{dt} \tag{3}$$

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:

$$\vec{a}_t \equiv (\vec{v}_{t+dt} - \vec{v}_t)/dt = 0 \tag{4}$$

We now solve eq. 2 and eq. 3 in $\vec{p}_{t+dt}$ and $\vec{v}_{t+dt}$ respectively. We find:

$$\vec{p}_{t+dt} = \vec{p}_t + \vec{v}_t dt \tag{5}$$
$$\vec{v}_{t+dt} = \vec{v}_t + \vec{a}_t dt \tag{6}$$

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

$$\vec{F}_t \propto \vec{a}_t \tag{7}$$

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

$$\vec{F}_t = m\vec{a}_t \tag{8}$$

We replace the solution of eq. 8 for a_t in eq. 6 and we obtain:

$$\vec{p}_{t+dt} = \vec{p}_t + \vec{v}_t dt \tag{9}$$
$$\vec{v}_{t+dt} = \vec{v}_t + (1/m)\vec{F}_t dt \tag{10}$$

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. 9 and 10 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.

So far we assumed m is 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:

$$\vec{K}_t \equiv m_t \vec{v}_t \tag{11}$$

In terms of $\vec{K}_t$, eq. 8 can be written as

$$\vec{F}_t = \frac{\vec{K}_{t+dt} - \vec{K}_t}{dt} \tag{12}$$

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

$$\vec{F}_t = \sum_i \vec{F}_i \quad \text{(compute force)} \tag{13}$$

$$\vec{v}_t = m_t^{-1} \vec{K}_t \quad \text{(compute velocity)} \tag{14}$$

$$\vec{p}_{t+dt} = \vec{p}_t + \vec{v}_t dt \quad \text{(update position)} \tag{15}$$

$$\vec{K}_{t+dt} = \vec{K}_t + \vec{F}_t dt \quad \text{(update momentum)} \tag{16}$$

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:

$$\vec{F} = \sum_i \vec{F}_i \tag{17}$$

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*.

4 Types of Forces and Newton Third Law

Here we are exclusively intersted in the following types of forces:

4.1 Gravity

$$\vec{F}_t^{gravity} \equiv (0, -m_t g, 0) \tag{18}$$

where $g = 9.8$meters/second.

4.2 Spring

$$\vec{F}_t^{spring} \equiv \kappa(|\vec{q}_t - \vec{p}_t| - L)\frac{\vec{q}_t - \vec{p}_q}{|\vec{q}_t - \vec{p}_t|} \tag{19}$$

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 spring. 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.

4.3 Friction

$$\vec{F}_t^{friction} \equiv -\gamma\vec{v}_t \tag{20}$$

γ is called the *friction coefficient*.

Newton's third law states that if a body A exercises a force F on a body B, then body B exercies a force $-F$ on body A. In any simulation we have to account for this using force generators that link objects to each other.

5 Rotations

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

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

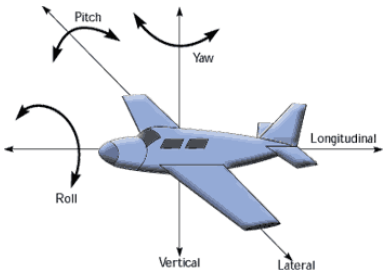
The matrix multiplication maps one vector p onto another vector p'. If we now require that the matrix R have determinant equal to 1 it follows that for every vector p

$$|\vec{p'}| = |R\vec{p}| = |R||\vec{p}| = |\vec{p}| \tag{22}$$

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:

$$\vec{p'} \cdot \vec{q'} = (R\vec{p}) \cdot (R\vec{q}) = \vec{p} \cdot \vec{q} \tag{23}$$

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

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

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

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

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

$$R_Z(\theta) \equiv \begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{26}$$

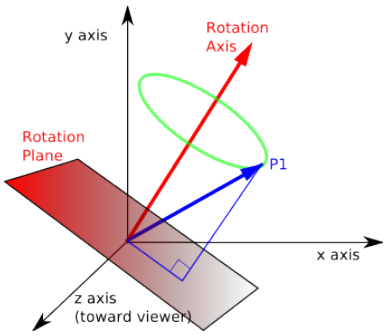
A rotation of the angle θ around an arbitrary direction $\hat{n} = (n_x, n_y, n_z)$ is given by:

$$R_{\hat{n}}(\theta) = \begin{pmatrix} tn_xn_x + c & tn_xn_y - sn_z & tn_xn_z + sn_y \\ tn_xn_y + sn_z & tn_yn_y + c & tn_yn_z - sn_x \\ tn_xn_z - sn_y & tn_yn_z + sn_x & tn_zn_z + c \end{pmatrix} \tag{27}$$

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

It is also conventinent to merge the directional information $\hat{n}$ with the angular information θ into a single vector $\vec{\theta} \equiv \theta\hat{n}$ and use the more compact notation:

$$R(\vec{\theta}) \equiv R_{\hat{n}}(\theta) \tag{28}$$



A rotation can also be used to represent the orientation of a body.

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

$$R(\vec{\theta'}) = R(\vec{\beta})R(\vec{\theta}) \tag{29}$$

notice that

$$\vec{\beta} = (\beta m_x, \beta m_y, \beta m_z) \tag{30}$$
$$\vec{\theta} = (\theta n_x, \theta n_y, \theta n_z) \tag{31}$$
$$\vec{\theta'} = (\theta' n'_x, \theta' n'_y, \theta' n'_z) \tag{32}$$

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

$$R' = R(\vec{\omega} dt)R \tag{33}$$

where

$$\vec{\omega} \equiv (\omega n_x, \omega n_y, \omega n_z) \tag{34}$$

is called angular velocity. What is the explicit expression for $R(\vec{\omega} dt)$ and first order in dt?

6 Spinning Objects

When a body is rotating with contant angular velocity $\vec{\omega}$, its rotation is proportional to time t only. We say the body is spinning. Here we consider body spinning around the direction $\hat{n} = (n_x, n_y, n_z)$ of an anglular velocity ω. The angular position of the object at time t is therefore $\theta_t = \omega t$. It is orinetation at time t is therefore given by:

$$\vec{\theta_t} \equiv (\omega t n_x, \omega t n_y, \omega t n_z) \tag{35}$$

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

$$\vec{p_t} = R(\vec{\theta_t})\vec{p} \tag{36}$$

and we can compute it velocity using the definition:

$$\begin{aligned} \vec{v_t} &= (\vec{p}_{t+dt} - \vec{p}_t)/dt & (37)\\ &= (R(\vec{\theta}_{t+dt}t)\vec{p} - R(\vec{\theta}_t)\vec{p})/dt & (38)\\ &= \vec{\omega} \times \vec{p} & (39) \end{aligned}$$

where $\vec{\omega}$ is the angular velocity, same as eq. 34.

7 Newton Second Law for Rotation

Before we have formulated the second law of Newton as

$$\vec{F_t} = \frac{\vec{K}_{t+dt} - \vec{K}_t}{dt} \tag{40}$$

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

$$\begin{aligned} \vec{r_t} \times \vec{F_t} &= \vec{r_t} \times \frac{\vec{K}_{t+dt} - \vec{K}_t}{dt} & (41)\\ &= \frac{\vec{r_t} \times \vec{K}_{t+dt} - \vec{r_t} \times \vec{K}_t}{dt} & (42) \end{aligned}$$

and if we define the *torque* as

$$\vec{\tau} \equiv \vec{r} \times \vec{F} \tag{43}$$

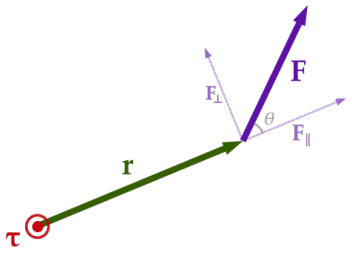
and the *angular momentum* as

$$\vec{L} \equiv \vec{r} \times \vec{K} = m\vec{r} \times \vec{v} \tag{44}$$

we can rewrite eq. 42 as

$$\vec{\tau}_t = \frac{L_{t+dt} - L_t}{dt} \tag{45}$$

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



Notice that if we substiture eq. 39 which gives the velocity in terms of the angualar velocity into eq. 44 we obtain:

$$\vec{L} = \vec{r} \times (m\vec{\omega} \times \vec{r}) = (m|\vec{r}|^2)\vec{\omega} \tag{46}$$

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

Let us now consider a rigid body comprised of many masses subject to constraints. For each mass m_i at point r_i we can write:

$$\begin{aligned}
\vec{\tau}_0 &= \vec{L}_0 && (47)\\
\vec{\tau}_1 &= \vec{L}_1 && (48)\\
\vec{\tau}_2 &= \vec{L}_2 && (49)\\
\ldots &\quad \ldots && (50)\\
\sum_i \vec{\tau}_i &= \sum_i \vec{L}_i && (51)\\
\vec{\tau} &= \vec{L} && (52)
\end{aligned}$$

In the last step we have used the following definitions:

$$\vec{\tau} \equiv \sum_i \vec{\tau}_i = \sum_i \vec{r}_i \times \vec{F}_i \tag{53}$$

and

$$\vec{L} \equiv \sum_i \vec{L}_i = \sum_i \vec{r}_i \times \vec{K}_i = m_i \vec{r}_i \times \vec{v}_i \tag{54}$$

Eq. 45 is still valid, but eq. 54 becomes:

$$\vec{L} = \sum_i \vec{r}_i \times (m_i \vec{\omega} \times \vec{r}_i) = I\vec{\omega} \tag{55}$$

where I is a 3x3 matrix, called *Inertia Tensor* with components:

$$I_{jk} \equiv \sum_i m_i(\delta_{jk}|\vec{r}_i|^2 - r_{i,j}r_{i,k}) \tag{56}$$

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:

$$\vec{F}_t = \sum_i \vec{F}_i \quad \text{(compute force)} \tag{57}$$

$$\vec{\tau}_t = \sum_i \vec{r}_i \times \vec{F}_i \quad \text{(compute torque)} \tag{58}$$

$$\vec{v}_t = m_t^{-1}\vec{K}_t \quad \text{(compute velocity)} \tag{59}$$

$$\vec{\omega}_t = I_t^{-1}L_t \quad \text{(compute angular velocity)} \tag{60}$$

$$\vec{p}_{t+dt} = \vec{p}_t + \vec{v}_t dt \quad \text{(update position)} \tag{61}$$

$$\vec{K}_{t+dt} = \vec{K}_t + \vec{F}_t dt \quad \text{(update momentum)} \tag{62}$$

$$R_t = R(\omega_t dt)R_t \quad \text{(update orientation)} \tag{63}$$

$$\vec{L}_{t+dt} = \vec{L}_t + \vec{\tau}_t dt \quad \text{(update angular momentum)} \tag{64}$$

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:

$$R_t\vec{r}_i + p_t \tag{65}$$

and the velocity of the point is

$$\vec{\omega}_t \times \vec{r}_i + \vec{v}_t \tag{66}$$

The position of a point $\vec{p}$ rotated by R around point $\vec{q}$ is given by:

$$\vec{p'} = R(\vec{p} - \vec{q}) + \vec{q} = R\vec{p} + (1 - R)\vec{q} \tag{67}$$

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:

$$m_{total} = \sum_i m_i \tag{68}$$

$$\vec{p}_{cm} = \sum \vec{p}_i m_i / m_{total} \tag{69}$$

$$I_{total} = \sum_i I_i + \Delta I(m_i, \vec{p}_i - \vec{p}_{total}) \tag{70}$$

where

$$\Delta I(m, \vec{r})_{jk} \equiv m(\delta_{jk}|\vec{r}|^2 - r_j r_k) \tag{71}$$

The object I is a 3×3 matrix and it is called *Inertia Tensor*. The inertia tensor depends on the orienttation of the object. If the object is rotated by R_t, the inertia tensor changes:

$$I_0 \to I_t = R_t I_0 R_t^{-1} \tag{72}$$

9 Quaternions

Quaterions provide a way to representant an orientation in 3D without using a rotation but using a 4-vector instead. They are not used in the code because they are not necessary.

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

$$\tilde{\theta} = (\theta_0, \theta_1, \theta_2, \theta_3) = (n_x \sin(\theta/2), n_y \sin(\theta/2), n_z \sin(\theta/2), \cos(\theta/2)) \tag{73}$$

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

$$R(\tilde{\theta}) \equiv \begin{pmatrix} 2(\theta_0\theta_0 + \theta_1\theta_1) - 1 & 2(\theta_0\theta_1 - \theta_2\theta_3) & 2(\theta_0\theta_2 - \theta_1\theta_3) \\ 2(\theta_0\theta_1 + \theta_2\theta_3) & 2(\theta_1\theta_1 + \theta_3\theta_3) - 1 & 2(\theta_1\theta_2 - \theta_0\theta_3) \\ 2(\theta_0\theta_3 - \theta_1\theta_3) & 2(\theta_0\theta_3 + \theta_1\theta_2) & 2(\theta_2\theta_2 + \theta_3\theta_3) - 1 \end{pmatrix} \tag{74}$$

The 4-vector $\tilde{\theta}$ defines an *orientation* and it is called a *quaternion* although its mathematical 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:

$$R(\tilde{\theta}') = R_{\hat{n}}(\beta)R(\tilde{\theta}) \tag{75}$$

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

$$\tilde{\theta}' = \tilde{\theta} + \frac{1}{2}\vec{\beta}\tilde{\theta} \tag{76}$$

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

$$\vec{\beta}\tilde{\theta} = \begin{pmatrix} -(\beta_0\theta_0 + \beta_1\theta_1 + \beta_2\theta_2) \\ (\beta_0\theta_3 + \beta_1\theta_2 - \beta_2\theta_1) \\ (\beta_1\theta_3 + \beta_2\theta_0 - \beta_0\theta_2) \\ (\beta_2\theta_3 + \beta_0\theta_1 - \beta_1\theta_0) \end{pmatrix} \tag{77}$$

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

Eq. 76 will come up handy later when talking about spinning objects. In this case for an infinitesimal rotation $\beta = \omega dt$:

$$\tilde{\theta}' = \tilde{\theta} + \frac{1}{2}\vec{\omega}\tilde{\theta}dt \tag{78}$$

10 Conservation Laws

The third Newton law says that for each force $\vec{F_i}$ there is an opposite force $\vec{F_j} = -F_i$ acting on a different body. This means that we add up all forces acting on all bodies:

$$\sum_i \vec{F_i} = \sum_i \mathrm{d}\vec{K}_i/\mathrm{d}t = 0 \tag{79}$$

(where we used the dot notation to indicate a derivative). This is equivalent to te statement that

$$\sum_i \vec{K}_i = constant \tag{80}$$

This is called *conservation of momentum*. It states that the sum of momenta K of all bodies has to be contant.

It can be proven (but we do not because it requires Lagrangian Mechanics) that for each inveriance of the system there is an associated conservation law.

Translation invariance implies *Consevration of momentum*:

$$\sum_i \vec{K}_i = constant \tag{81}$$

Rotation invariance implies *Conservation of angular momentum*:

$$\sum_i \vec{L}_i = constant \tag{82}$$

Time translation invariance implies *Conservation of energy*:

$$\sum_i E_i = constant \tag{83}$$

where E_i, the energy of body i and it equal to:

$$E_i = \frac{1}{2}mv_i^2 + \frac{1}{2}I\omega_i^2 - \int_{t_0}^{t} \vec{F}_i(t) \cdot \vec{v}_i(t)dt - \int_{t0}^{t} \vec{\tau}_i(t) \cdot \vec{\omega}_i(t)dt \tag{84}$$

The difference pieces are called respectively: kinetic energy, rotational kinetic energy, poetntial energy, and rotational potential energy. They measure respectively: how fast the body moves, how fast it spins, how much work was done by the forces acting upon it, and how much work was done by the torques accting upon it. t_0 is a time that one can start arbitrarily and by convention.

11 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 closing velocity is defined as:

$$\vec{v}_{closing} \equiv \vec{v}_B - \vec{v}_A \tag{85}$$

The separating velocity (relative velocity after the collision) is defined as:

$$\vec{v}_{separating} \equiv \vec{v}'_B - \vec{v}'_A \tag{86}$$

The separating velocity can be written in terms of the closing velocity:

$$\vec{v}_{separating} = -c\vec{v}_{closing} \tag{87}$$

where c is a restitution coefficient. For an elastic collision $c = 1$ and for an inelastic collision $c = 0$. Conservation of momentum and the above relation between closing and separating velocity yields:

$$\vec{K}'_A = \vec{K}_A + \vec{J} \tag{88}$$

$$\vec{K}'_B = \vec{K}_B - \vec{J} \tag{89}$$

where

$$\vec{J} = \frac{(c+1)\vec{v}_{closing} \cdot \hat{n}}{1/m_A + 1/m_B}\hat{n} \tag{90}$$

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

$$\vec{K}'_A = \vec{K}_A + \lim_{m_A \to \infty} + \vec{J} = \vec{K}_A + m_A(c+1)(\vec{v}_{closing} \cdot \hat{n})\hat{n} \tag{91}$$

When the $v_{closing}$ is in the same direction as n then:

$$\vec{K}'_A = \vec{K}_A + m_A(c+1)\vec{v}_{closing} \tag{92}$$

In the case of rigid bodies eq. 87 applies to the points of contact between two rigid bodies. The collision is completed determined by the collision point $\vec{q}$ and the normal to the collision plane $\hat{n}$.

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.

$$\vec{J}_\perp = \frac{(c+1)\vec{v}_{closing}) \cdot \hat{n}}{1/m_A + 1/m_B + [(1/I_A)(\vec{r}_{cA} \times \hat{n}) \times r_{cA} + (1/I_B)(\vec{r}_{cB} \times \hat{n}) \times r_{rB}] \cdot \hat{n}}\hat{n} \tag{93}$$

Here $\vec{r}_{cA} = \vec{q} - \vec{p}_A$ is the point of collision respect to the center of mass $\vec{p}_A$, $\vec{v}_{cA} = \vec{\omega}_A \times \vec{r}_A + \vec{v}_A$ 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. Similarly, $\vec{r}_{cB} = \vec{q} - \vec{p}_B$ and $\vec{v}_{cB} = \vec{\omega}_B \times \vec{r}_B + \vec{v}_B$.

Once this impulse 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 impulse to take into account:

$$\vec{J}_{\parallel} = \frac{(c_f + 1)\vec{v}_{closing} \cdot \hat{t}}{1/m_A + 1/m_B + \left[(1/I_A)(\vec{r}_{cA} \times \hat{t}) \times r_{cA} + (1/I_B)(\vec{r}_{cB} \times \hat{t}) \times r_{rB}\right] \cdot \hat{t}} \hat{t} \tag{94}$$

where $\hat{t}$ is a versor orthogonal to n and to the relative velocity beween the colliding points:

$$\hat{t} = \frac{\hat{n} \times \vec{v}_{closing}}{|\hat{n} \times \vec{v}_{closing}|} \tag{95}$$

In general c_f is different from c.

Given the total inpulse $\vec{J} = \vec{J}_{\perp} + \vec{J}_{\parallel}$, we can resolve the collisiong by correcting the momenta and angular momenta:

$$\vec{K}'_A = K_A + \vec{J} \tag{96}$$
$$\vec{\tau}'_A = \tau_A + \vec{r}_{cA} \times \vec{J} \tag{97}$$
$$\vec{K}'_B = K_B - \vec{J} \tag{98}$$
$$\vec{\tau}'_B = \tau_B - \vec{r}_{cB} \times \vec{J} \tag{99}$$

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. 94)
- update the force and torque to include one time contribution of the impulse (eq. 99).

12 Geometry of collision

Here we are concerned with the problem of terminating if two objects have collided. If the objects are made of vertices and faces the possible collisions can be vertex-vertex, vertex-edge, verted-face, edge-face, face-face. Most of them can be ignored but vertex-face and edge-edge.

Let's focus on vertex-face collision. In order to resolve the collisions we need to:

- detect a collision has occurred;
- find the collision point, $\vec{q}$;
- find the normal to the collision plane, $\hat{n}$;
- resolve as in previous section.

We will assume the body is convex (the center of mass is inside the body).

We will label $\vec{p}_A$ the center of mass of body A, $\vec{p}_B$ the center of mass of body B, $\vec{q}$ the collision point, $\hat{n}$ the normal to the collision point, $\vec{f_1}$,$\vec{f_2}$,$\vec{f_3}$ the vertices of the face of A and $\vec{u}$ the vertex of body B which may have hit the face of object A.

First we compute $\vec{n}$:

$$\vec{n} = (\vec{f_2} - \vec{f_1}) \times (\vec{f_3} - \vec{f_1}) \tag{100}$$

and normalized it $\hat{n} = \vec{n}/n$. Than we compute q:

$$\vec{q} = \vec{u} - (\vec{u} \cdot \hat{n} - \vec{f_1} \cdot \hat{n})\hat{n} \tag{101}$$

And we check that $\vec{q}$ lays inside the face:

$$|(\vec{q} - \vec{f_1}) \times (\vec{f_2} - \vec{f_1})| \quad + \tag{102}$$
$$|(\vec{q} - \vec{f_2}) \times (\vec{f_3} - \vec{f_2})| \quad + \tag{103}$$
$$|(\vec{q} - \vec{f_3}) \times (\vec{f_1} - \vec{f_3})| \quad <= \quad |(\vec{f_1} - \vec{f_2}) \times (\vec{f_3} - \vec{f_1})| \tag{104}$$

If this is the case, we check the the projection along $\hat{n}$ of the points $\hat{p}_A$ (the center of mass of A), the vertex $\hat{v}$, the collision point $\vec{q}$ and $\vec{p}_B$ (the center of mass of B) and in proper spatial order:

$$((\vec{p}_A - \vec{p}_B) \cdot \hat{n})((\vec{u} - \vec{q}) \cdot \hat{n}) >= 0 \tag{105}$$

If the collision occurs the penetration is $\vec{p} - \vec{q}$.

Finally we loop over all faces of A, and vertices of B, we run the above algorithm and if a collision occurs we resolve the penetration, we compute the velocity of the collision point and we apply the impulse at collision point as discussed in the previous section.

13 Implementation

```
// Program name: cylon.cpp
// Author:       Massimo Di Pierro
// License:      BSD
```

Import the necessary libraries:

```
#include "fstream"
#include <sstream>
#include <string>
#include <iostream>
#include "math.h"
#include "stdlib.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 self (*this)
#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 Pi = 3.1415926535897931;
const float PRECISION = 0.00001;
const int X = 0;
const int Y = 1;
const int Z = 2;
const float DT  = 0.017f; // Hard-coded dt for Universe::evolve()
const int MSPF = 17; // msec per frame: glutTimerFunc() only takes integers
    for params
                     // lower value = higher frame rate


float uniform(float a=0, float b=1) {
  int n = 10000;
  return a+(b-a)*(float)(rand() % n)/n;
}
```

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]; }
  Matrix t();
};
```

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;
}

Matrix Matrix::t() {
  Matrix Mt;
  forXYZ(j) forXYZ(k) Mt(j,k)=self(k,j);
  return Mt;
}
```

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;
  array<Vector> r;   // vertices in local coordinates
  array<array<int> > faces;
  Vector color;      // color of the object;
  bool visible;
  // 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 /////////////////////////////
  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, float radius=0.2, bool locked=false) {
    this->m = m;
    this->radius = radius;
    this->locked = locked;
    this->R = Rotation();
    I(X,X)=I(Y,Y)=I(Z,Z)=m;
    this->inv_I = 1.0/I;
    this->r.push_back(Vector(0,0,0)); // object contains one point
    this->color = Vector(1,0,0);
    this->visible = true;
    this->update_vertices();
  }
  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() {
  if(Rr.size()!=r.size()) Rr.resize(r.size());
  if(vertices.size()!=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     = (1.0/m)*K;
  omega = R*(inv_I*(R.t()*L));      // R*inv_I*R.t() in rotated frame
  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 = ((iB<0)?(bodyB->p):(bodyB->vertices[iB]))-
      ((iA<0)?(bodyA->p):(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;
      if(iA>=0)
        bodyA->tau = bodyA->tau + cross(bodyA->Rr[iA],F);
      if(iB>=0)
        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 = pin - ((i<0)?(body->p):(body->vertices[i]));
    float n = norm(d);
    Vector F = kappa*(n-L)*d/n;
    body->F = body->F+F;
    if(i>=0)
      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) {
      this->body = body; this->gamma = gamma;
    }
    void apply(float dt) {
      body->F = body->F-gamma*(body->v)*dt; // ignores shape
    }
};
```

Class water, one instance floods the universe:

```
class Water: public Force {
public:
  float level, wave, speed;
  float m[41][41];
  float t, x0,dx;
  set<Body*> *bodies;
  Water(set<Body*> *bodies, float level,
        float wave=0.2, float speed=0.2) {
    this->bodies = bodies;
    this->level = level; this->wave = wave, this->speed = speed;
    t = 0; x0 = 5.0; dx = 0.25;
  }
  void apply(float dt) {
```

```
    set<Body*>::iterator ibody;
    t = t+dt;
    for(int i=0; i<41; i++)
      for(int j=0; j<41; j++)
        this->m[i][j] = level+wave/2*sin(speed*t+i)+wave/2*sin(0.5*i*j);
    forEach(ibody,*bodies) {
      Body &body = OBJ(ibody); // dereference
      int i = (body.p(X)+x0)/dx;
      int j = (body.p(Z)+x0)/dx;
      if(body.p(Y)<m[i][j]) {
        body.F = body.F + (Vector(0,1,0)-2.0*(body.v))*dt;
        body.L = (1.0-dt)*body.L;
      }
    }
  }
  void draw();
};

Vector resolve_collision(Body &A, Body &B, Vector q, Vector n,
             float c, float cf) {
  Vector r_cA = q-A.p;
  Vector r_cB = q-B.p;
  Vector v_cA = cross(A.omega,r_cA)+A.v;
  Vector v_cB = cross(B.omega,r_cB)+B.v;
  Vector v_closing = v_cB-v_cA;
  Vector t = versor(cross(n,v_closing));
  Vector Jo,Jp,J;
  Jo = (c+1)*(v_closing*n)/
    (1.0/A.m+1.0/B.m+(A.inv_I*cross(cross(r_cA,n),r_cA)+
              B.inv_I*cross(cross(r_cB,n),r_cB))*n)*n;
  Jp = (cf+1)*(v_closing*t)/
    (1.0/A.m+1.0/B.m+(A.inv_I*cross(cross(r_cA,t),r_cA)+
              B.inv_I*cross(cross(r_cB,t),r_cB))*t)*t;
  J = Jo+Jp;
  A.K = A.K + J;
  B.K = B.K - J;
  A.L = A.L + cross(r_cA,J);
  B.L = B.L - cross(r_cB,J);
}
```

A constraint has detect and resolve methods:

```
class Constraint {
public:
  virtual bool detect()=0;
  virtual void resolve(float dt)=0;
  virtual void draw() {}
};
```

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 resolve(float dt) {
    set<Body*>::iterator ibodyA, ibodyB;
    forEach(ibodyA,*bodies) {
      forEach(ibodyB,*bodies) {
    if((*ibodyA)<(*ibodyB)) {
      Body &A = OBJ(ibodyA); // dereference
      Body &B = OBJ(ibodyB); // dereference
      Vector d = B.p-A.p;
      Vector n = d/norm(d);
      Vector v_closing = B.v-A.v;
      float penetration = A.radius+B.radius-norm(d);
      if(penetration>0 && v_closing*n<0) {
        // move bodies to eliminate penetration
        Vector delta = (penetration/(A.m+B.m))*versor(d);
        A.p = A.p-B.m*delta;
        B.p = B.p+A.m*delta;
        // compute cotact point
        Vector n = versor(B.p-A.p);
        Vector q = A.p + A.radius*n;
        resolve_collision(A,B,q,n,restitution,restitution);
      }
        }
      }
```

```
    }
  }
};
```

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() { bodies.insert(new Body()); };
  virtual void callback() {};
  virtual void mouse(int button, int state, int x, int y) {};
  virtual void keyboard(unsigned char key, int x, int y) {};
};
```

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.color = 0.5*(a.color+b.color);
  c.radius = max(a.radius+norm(a.p-c.p),b.radius+norm(b.p-c.p));
  c.R = Rotation();
  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+cross(a.p-c.p,a.K)+b.L+cross(b.p-c.p,b.K);
  Vector da = a.p-c.p;
  Vector db = b.p-c.p;
  int na = a.r.size();
  int nb = b.r.size();
  c.I(X,X) = c.I(Y,Y)= c.I(Z,Z) =0;
  // copy all r
  Vector v;
  c.r.resize(na+nb);
  for(int i=0; i<na; i++) {
    v = a.R*a.r[i]+da;
    c.r[i] = v;
    c.I = c.I + dI(a.m/na,v);
  }
  for(int i=0; i<nb; i++) {
    v = b.R*b.r[i]+db;
    c.r[i+na] = v;
    c.I = c.I + dI(b.m/nb,v);
  }
  // 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]+na);
  c.inv_I = 1.0/c.I;
  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;
  r.resize(0);
  faces.resize(0);
  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 a universe with an airplane:

```
class MyUniverseAirplane : public Universe {
  Body plane;
public:
  void build_universe() {
    plane.color=Vector(1,0,0); //red
    plane.loadObj("assets/plane.obj",0.5);
    plane.R = Rotation(Vector(0,Pi,0));
    bodies.insert(&plane);
    forces.insert(new GravityForce(&plane,0.5));
    // constraints.insert(new PlaneConstraint(&plane,0.0,Vector(0,-0.2,0),
        Vector(0,-1,0)));
  }
  void callback() {
  }
  void keyboard(unsigned char key, int x, int y) {
    if(key=='w') plane.L = plane.L + Vector(+0.1,0,0);
    if(key=='a') plane.L = plane.L + Vector(0,0,+0.1);
    if(key=='d') plane.L = plane.L + Vector(0,0,-0.1);
    if(key=='z') plane.L = plane.L + Vector(-0.1,0,0);
    if(key=='n') plane.K = plane.K + Vector(0,0,+1);
    if(key=='m') plane.K = plane.K + Vector(0,0,-1);
  }
};


array<float> random_vector(int n, float M) {
  float one_norm = 0.0;
  array<float> m(n);
  for(int i=0; i<n; i++) { m[i] = uniform(); one_norm += m[i]; }
  for(int i=0; i<n; i++) m[i]=M*m[i]/one_norm;
  return m;
}
```

Make a universe with an exploding body:

```
class SimpleUniverse : public Universe {
private:
  Body* xb;
public:
  void build_universe() {
    float m = 1.0;
    float v = 5.0;
    float theta = Pi/2.2;
    Body &b = *new Body(m);
    b.color = Vector(uniform(),uniform(),uniform()); //red
    b.loadObj("assets/sphere.obj",0.5);
    b.p = Vector(0,10,-10);
    b.K = Vector(0,0,0);
    b.L = Vector(0,0,0);
    bodies.insert(&b);
    forces.insert(new GravityForce(&b,1.0));
    xb = &b;
  }
  void callback() {
    if(frame==150) {
      int n = 50;
      float vx,vy,vz;
      float E = 0.5;
      float mysum_x=0, mysum_y=0, mysum_z=0;
      array<float> m = random_vector(n,xb->m);
      array<float> eps_x(n), eps_y(n), eps_z(n);
      xb->visible = false;
      for(int j=0; j<n; j++) {
        cout << j << "  " << uniform() << endl;
        Body &a = *new Body(m[j]);
        a.color = Vector(uniform(),uniform(),uniform());
        a.p = Vector(xb->p(X), xb->p(Y), xb->p(Z));
        if(j<n-1) {
          eps_x[j] = uniform(-1,1)*sqrt(E);
          eps_y[j] = uniform(-1,1)*sqrt(E);
          eps_z[j] = uniform(-1,1)*sqrt(E);
          mysum_x += m[j]*eps_x[j];
          mysum_y += m[j]*eps_y[j];
          mysum_z += m[j]*eps_z[j];
        } else {
          eps_x[j] = -mysum_x/m[j];
          eps_y[j] = -mysum_y/m[j];
          eps_z[j] = -mysum_z/m[j];
        }
        vx = xb->v(X) + eps_x[j];
        vy = xb->v(Y) + eps_y[j];
        vz = xb->v(Z) + eps_z[j];
        a.K = Vector(m[j]*vx, m[j]*vy, m[j]*vz);
        bodies.insert(&a);
        forces.insert(new GravityForce(&a,1.0));
      }
    }

    float restitution = 0.5;
    forEach(ibody,bodies)

      if(OBJ(ibody).p(Y)<0 && OBJ(ibody).K(Y)<0) {
```

```
        OBJ(ibody).p(Y)=-restitution*OBJ(ibody).p(Y);
        OBJ(ibody).K(Y)=-restitution*OBJ(ibody).K(Y);
      }
  }
};
```

Make a universe with some bouncing balls:

```
class MyUniverse : public Universe {
public:
  void build_universe() {
    Body *b_old = 0;
    for(int i=0; i<30; i++) {
      Body &b = *new Body();
      b.color=Vector(uniform(0,1),uniform(0,1),uniform(0,1));
      b.loadObj("assets/sphere.obj");
      b.p = Vector(uniform(-2,2),uniform(1,3),uniform(-2,2));
      b.K = Vector(uniform(-2,2),uniform(-2,2),0);
      b.L = Vector(uniform(0,1),uniform(0,1),uniform(0,1));
      bodies.insert(&b);
      forces.insert(new GravityForce(&b,1));
      constraints.insert(new PlaneConstraint(&b,0.9,Vector(0,0,0),
                                             Vector(0,-1,0)));
      constraints.insert(new PlaneConstraint(&b,0.9,Vector(0,5,0),
                                             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)));
    }
    constraints.insert(new All2AllCollisions(&bodies,0.8));
    // forces.insert(new Water(&bodies,3.0));
  }
};
```

Make a universe with composite objects made of cubes:

```
class MyUniverseCubes : public Universe {
public:
  Body cube(const Vector &p, const Vector& theta, const Vector &color) {
    Body cube;
    cube.color = color;
    cube.p = p;
    cube.R = Rotation(theta);
    for(int x=-1; x<=+1; x+=2)
      for(int y=-1; y<=+1; y+=2)
        for(int z=-1; z<=+1; z+=2)
          cube.r.push_back(Vector(x,y,z));
    cube.faces.resize(6);
    for(int i=0; i<2; i++) {
      cube.faces[i].push_back(0+4*i);
      cube.faces[i].push_back(1+4*i);
      cube.faces[i].push_back(3+4*i);
      cube.faces[i].push_back(2+4*i);
```

```
      cube.faces[i+2].push_back(0+2*i);
      cube.faces[i+2].push_back(1+2*i);
      cube.faces[i+2].push_back(5+2*i);
      cube.faces[i+2].push_back(4+2*i);
      cube.faces[i+4].push_back(0+i);
      cube.faces[i+4].push_back(2+i);
      cube.faces[i+4].push_back(6+i);
      cube.faces[i+4].push_back(4+i);
    }
    cube.update_vertices();
    return cube;
  }
  void build_universe() {
    Body *thing = new Body();
    (*thing) =
      cube(Vector(0,4,0),Vector(0,1,0),Vector(0,1,0)) +
      cube(Vector(1,4,0),Vector(1,0,0),Vector(0,1,0.2)) +
      cube(Vector(1,4.5,.5),Vector(1,0,1),Vector(0.2,1));
    (*thing).L = Vector(0,0.1,0);
    bodies.insert(thing);
    Body *other_thing = new Body();
    (*other_thing) = (*thing);
    (*other_thing).p = Vector(-1,3,0);
    (*other_thing).L = Vector(-0.1,0.0,0.1);
    bodies.insert(other_thing);
  }
};

class CompositionUniverse: public Universe {
private:
  Body b;
public:
  void build_universe() {
    Body a;
    float x,y,z;
    for(int i=0; i<8; i++) {
      x = 2*((i>>0)&1)-1;
      y = 2*((i>>1)&1)-1;
      z = 2*((i>>2)&1)-1;
      a.p = Vector(x,y+1,z-5);
      if(i==0) b = a; else b=b+a;
    }
    b.L = Vector(5,10,0);
    bodies.insert(&b);
  }
};

MyUniverse universe;
// MyUniverseAirplane universe;
// MyUniverseCubes universe;
// SimpleUniverse universe;
// CompositionUniverse 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(int value) {
  // evolve world
  universe.dt = DT;
  universe.evolve();
  glutTimerFunc(MSPF, update, 0);
  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)
    if(OBJ(universe.ibody).visible)
                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);
  if(faces.size()==0) {
    glColor3f(color(0),color(1),color(2));
    int n = vertices.size();
    for(int i=0; i<n; i++) {
      glPushMatrix();
      glTranslatef(vertices[i].v[X],vertices[i].v[Y],vertices[i].v[Z]);
      glutSolidSphere(0.2,10,10);
      glPopMatrix();
    }
  } else
    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);
  if(iA<0)  glVertex3fv(bodyA->p.v);
  else glVertex3fv(bodyA->vertices[iA].v);
  if(iB<0)  glVertex3fv(bodyB->p.v);
  else 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) {
  universe.mouse(button,state,x,y);
}
```

Function called when a key is pressed:

```
void keyboard(unsigned char key, int x, int y) {
  universe.keyboard(key,x,y);
}
```

Called when the mouse is dragged:

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

The main function. everythign starts here:

```
1  int main(int argc, char** argv) {
2    // Create the application and its window
3    glutInit(&argc, argv);
4    createWindow("Cylon");
5    // fill universe with stuff
6    universe.build_universe();
7    // Set up the appropriate handler functions
8    glutReshapeFunc(reshape);
9    glutKeyboardFunc(keyboard);
10   glutDisplayFunc(display);
11   glutMouseFunc(mouse);
12   glutMotionFunc(motion);
13   glutTimerFunc(MSPF, update, 0); // Do not refresh faster than target
         framerate
14   // Enter into a loop
15   glutMainLoop();
16   return 0;
17 }
```