Workshop Series: Rigid Body Dynamics

Benjamin Kenwright[1]*

Abstract
The dynamics of a rigid body system is defined by its equations of motion. Rotational motion is more complicated than linear motion, and only the motion of rigid bodies will be considered here. Combining both rotational and linear motion to our physics simulator allows us to create full six degrees of freedom motion. This section combines the principles of rotational and translational movement using numerical integration to resolve Newtonian mechanics in an iterative manner.

Keywords
Torque, Rigid Body, Angular Velocity, Inertia, Integration, Euler, Verlet, Damping, Physics, Numerical Integration, Real-Time, Video Games, Interactive, Classical Mechanics, C++

[1] *Workshop Series (www.xbdev.net) - Benjamin Kenwright*

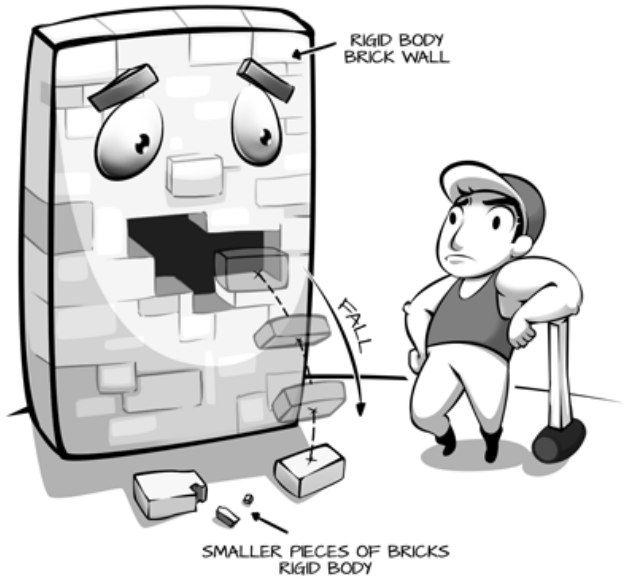

Figure 1. Rigid bodies - Rigid bodies enable your simulator objects to act under the control of physics. A rigid body can receive forces and torque to make your objects move in a realistic way. Any simulation object must contain a *rigid body* to be influenced by gravity, act under forces, or interact with other objects through the physics engine.

Contents

Introduction

Overview Our physics simulation is now capable of moving objects around; however, that movement probably looks rather unconvincing for anything other than particles. Larger objects, such as balls, crates, asteroids, and robots will behave a lot more believably if they rotate as they move. When you throw a beach-ball, a banana, or a brick, it doesn't simply move through the air at exactly the orientation that it was in when it left your hand, it rotates. When it subsequently bounces off something, it rotates at a different speed around a different axis. In this practical, we will apply ideas from previous practicals for numerical integration to the laws of angular motion.

Essentials In the earlier practicals, we discussed the concept of rigid bodies and distinguished them from a particle-based system by the introduction of angular velocity and orientation. As rigid bodies have some volume (i.e., they are a constant shape and size), their orientation in space is

important and must be simulated by the physics simulation. Before we discuss how this is implemented, we will take a quick refresher course on what angular motion is, and how it is represented mathematically.

1. Angular Mathematics

Principles and Concepts The orientation of an object, and how quickly that orientation changes, are modelled through angular mathematics. The concept is pretty much identical to that of linear motion, so we have direct counter-parts to position, velocity, and acceleration. These are the angle or orientation θ, the angular velocity ω, and the angular acceleration α. As with linear motion, these are three dimensional parameters represented by vectors, with each member containing the angular information about the x, y and z axes respectively.

Relationship The relationship between angle, angular velocity, and angular acceleration is the same as for their linear equivalents, in that the angular velocity is the rate of change of the angle over time, and the angular acceleration is the rate of change of angular velocity over time as given by Equation 1 below:

$$\begin{aligned} \omega &= \frac{\theta}{dt} \\ \alpha &= \frac{\omega}{dt} \end{aligned} \tag{1}$$

Conversely, angular velocity is the integral of angular acceleration over time, and angle is the integral of angular velocity over time, as given by Equation 2 below:

$$\begin{aligned} \omega &= \int \alpha \, dt \\ \theta &= \int \omega \, dt \end{aligned} \tag{2}$$

Clearly, we can use the same numerical integration techniques to solve these equations as we used for linear motion in the previous practical. In order to achieve this, we need to be able to calculate the angular acceleration of our simulated objects from the forces acting upon them.

2. Radians

The basic unit for angular motion is the *Radian*. While it is possible to store rotations as either degrees or rotations, it is much more straightforward, and consistent, to use radians. If you need to use trigonometric functions, such as sine and cosine, they will expect the parameters to be given in radians. An angle expressed in radians is defined as the ratio of the arc length s swept out by the angle θ, to the radius of the corresponding circle r.

$$\theta = \frac{s}{r} \tag{3}$$

Consequently, a full rotation of 360 deg is expressed as 2π radians; half of a complete revolution (i.e., 180 deg) is π radians and so on. In most physics simulations, if an angle gets bigger than 2π radians then it is reduced by 2π radians. This has no effect on the mathematics, or the simulation, as the orientation is identical (i.e., if something has rotated by 2π radians, then it is back at the original orientation of zero radians). To give an example, imagine the wheel on a car in a racing game - as the car moves along the track, the wheel rotates repeatedly; if we don't reset the orientation every revolution, then the angle will quickly become extremely high with no additional benefit or accuracy. This is achieved in C++ as shown below in Listing 1:

Listing 1. Wrapping Angular Values

```
1  if ( curAngle > TWOPI2 ) curAngle -= TWOPI;
2  if ( curAngle < TWOPI2 ) curAngle += TWOPI;
```

Note, that a defined constant is used (TWOPI) in Listing 1, instead of multiplying π by 2.0 for every object. Furthermore, note that this is only checked for the angle - it is acceptable for an angular velocity or even acceleration to exceed 2π, if something is spinning very fast.

3. Torque and Inertia

As with the linear motion of the objects simulated by the physics engine, we need to relate the angular acceleration of a rigid body to the forces acting upon it. This is achieved through the use of Torque (τ) and Moment of Inertia, which can be thought of as the angular equivalents of Force and Mass. The relationship between the torque τ acting on a body to the inertia I of that body and the resulting angular acceleration α is given by Equation 4:

$$\tau = I \, \alpha \tag{4}$$

This is the rotational equivalent to Newton's second law for linear motion $F = ma$.

The moment of inertia of a rigid body represents the amount of resistance a body has to changing its state of rotational motion (i.e., its angular acceleration). The moment of inertia depends on how the mass is distributed about the axis. For a given total mass, the moment of inertia is greater if more mass is farther from the axis than if the same mass is distributed closer to the axis. The classic example of this is the ice dancer who brings her arms vertically above her body to increase the speed at which she is spinning, or holds her arms out horizontally to slow down her spinning speed.

As we are simulating three-dimensional worlds, a scalar value of I does not contain sufficient information to describe the inertial behaviour of a body - the body's shape and the axis of rotation have an effect on its behaviour. We therefore introduce the concept of the Inertia Matrix or Inertia Tensor. First, we'll expand our equation for angular acceleration to express the torque and acceleration as vectors, and the inertia tensor as a matrix:

$$\begin{bmatrix} \tau_x \\ \tau_y \\ \tau_z \end{bmatrix} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} \begin{bmatrix} \alpha_x \\ \alpha_y \\ \alpha_z \end{bmatrix} \tag{5}$$

Each element of the inertia matrix represents the effect a torque around a particular axis has on the acceleration around a particular axis. So I_{xx} represents the effect that a torque around the x axis has on acceleration around the x axis, I_{xy} represents the effect that a torque around the x axis has on acceleration around the y axis, etc. For a completely symmetrical object, a torque around the x axis will cause acceleration around the x axis only; however, for more complex objects a torque around a particular axis may cause acceleration around the other two axes. For example, an evenly weighted cube floating in space is a completely symmetrical object. Applying a rotational nudge around its x axis will cause it to spin around its x axis only; the angular velocity around its y and z axes will remain zero. However, if we attach a very heavy lump to one of the cube's corners, then this will affect how it spins - a nudge around the unevenly weighted cube's x axis will cause a wobble around the other two axes, as the inertia of the heavy lump drags the cube off its rotation. As a further example, reconsider the spinning skater with her arms held out horizontally - if you could attach a heavy weight to one of her arms, she would wobble over and fall almost immediately, the poor thing.

Before we move on, there are a couple of important properties of the inertia matrix that we will discuss in a little more detail:

- The diagonal elements (I_{xx}, I_{yy}, I_{zz}) of the matrix must 'not' be zero
- The matrix must be symmetrical, that is $I_{xy} = I_{yx}$, $I_{xz} = I_{zx}$, and $I_{zy} = I_{yz}$

4. Symmetrical Objects

Multiple elements in a game can be simulated by the physics system as completely symmetrical objects - i.e. the objects' weights are evenly distributed around the their centre of gravity - even many objects which, in the real world, would not actually be symmetrical, such as barrels, bricks, and crates.

As we have discussed, a completely symmetrical object only rotates around the axes which have torque applied to them. This behaviour is defined in the inertia matrix by ensuring that the non-diagonal elements are set to zero. So the inertia matrix for a symmetrical object takes the form:

$$\begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \tag{6}$$

The diagonal elements must be non-zero. To understand why, consider the equation for the x axis torque:

$$\tau_x = I_{xx}\,\alpha_x \tag{7}$$

This equation defines the amount of torque required to instigate an angular acceleration around the x axis. The lower the value of I_{xx}, the higher the angular acceleration α_x that is produced by a particular amount of torque τ_x. If the inertia I_{xx} is zero, then an infinite angular acceleration would be produced by that same value of torque. This is clearly undesirable, and indeed physically impossible, which is why the diagonal elements of the inertia tensor must be non-zero. If your physics engine is exhibiting weird behaviour for symmetrical objects, then you should test that the diagonal elements are non-zero, and that the non-diagonal elements are zero.

5. Non-Symmetrical Objects

In the real world, almost no objects are symmetrical. However, in practice, e.g., virtual gaming environments, simulating non-symmetrical objects is computationally expensive compared to simulating symmetrical objects, so care should be taken that the extra computation involved is actually worthwhile for the intended effect. Simulating the rotational behaviour of non-symmetrical bodies requires the impulse matrix to be fully populated - i.e., some of the non-diagonal elements are not zero.

The diagonal elements must be non-zero, for the same reasons as described above for symmetrical bodies (i.e. a zero element in the diagonal will result in errors related to an infinite angular acceleration).

Mathematically, the elements of the inertia matrix are calculated by considering the rigid body as a continuous set of connected particles, in fixed positions relative to one another, and integrating their momentum across the volume of the body. We won't go into the details of this here, but the equations for calculating the diagonal elements are:

$$\begin{aligned} I_{xx} &= M \int_V (y^2 + z^2)\, dV \\ I_{yy} &= M \int_V (x^2 + z^2)\, dV \\ I_{zz} &= M \int_V (x^2 + y^2)\, dV \end{aligned} \tag{8}$$

and the non-diagonal elements are:

$$\begin{aligned} I_{xy} &= -M \int_V (xy)\, dV & I_{yx} &= -M \int_V (yx)\, dV \\ I_{xz} &= -M \int_V (zy)\, dV & I_{zx} &= -M \int_V (zx)\, dV \\ I_{yz} &= -M \int_V (yz)\, dV & I_{zy} &= -M \int_V (zy)\, dV \end{aligned} \tag{9}$$

It can be seen that the equations for I_{xy} and I_{yx} are equivalent. Similarly, for the other two pairs of diametrically oppo-

site elements. Hence, the inertia matrix must be symmetrical for meaningful rotational simulation. If your physics engine is exhibiting weird behaviour for non-symmetrical objects, then you should test that the inertia matrix is symmetrical, and that the diagonal elements are non-zero.

6. Inertial Matrix for Cuboid and Spherical Objects

The majority of objects simulated in your physics engine can be represented either as a solid sphere, or as a solid cuboid, so we will look at how to calculate the inertia matrix for such shapes. Remember, that the rendered shapes of the graphical objects are unlikely to be perfect spheres or cuboids, but for the purposes of physical simulation, most game objects can be represented by one or more basic shapes of this type. For example, a girder falling from a collapsing building can be simulated in the physics simulation as a long cuboid, while a roughly hewn asteroid can be simulated by a sphere. This idea of having two representations of a game object (the graphical shape which is rendered, and the physical shape which is simulated) is central to game development, and will be discussed in much more detail when we move on to collision detection and response.

6.1 Solid Sphere

The equation for calculating the inertia of a solid sphere of radius r and mass m is:

$$I = \frac{2mr^2}{5} \tag{10}$$

and the inertia matrix is constructed from setting the diagonal elements equal to this value, and the non-diagonal elements equal to zero.

$$\begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{bmatrix} \tag{11}$$

Hence, the sphere rotates around each axis equally, with no non-symmetrical behaviour; the matrix is symmetrical and the diagonal values are non-zero. If for some reason, you require your sphere to rotate more easily about a specific axis, then the value of I for that axis should be reduced slightly; and for a stiffer rotation response, the value should be increased somewhat.

6.2 Solid Cuboid

The equations for calculating the inertia of a solid cuboid of length l, height h, width w and mass m are:

$$\begin{aligned} I_{xx} &= \frac{1}{12} m(h^2 + w^2) \\ I_{yy} &= \frac{1}{12} m(l^2 + w^2) \\ I_{zz} &= \frac{1}{12} m(h^2 + l^2) \end{aligned} \tag{12}$$

again the inertia matrix is constructed from setting the diagonal elements equal to these values, and the non-diagonal elements equal to zero.

$$\begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} \tag{13}$$

So the cuboid rotates around each axis according to how big it is along that axis, with no non-symmetrical behaviour; the matrix is symmetrical and the diagonal values are non-zero. If for some reason, you require your cuboid to rotate more easily about a specific axis, then the value of I for that axis should be reduced slightly; and for a stiffer rotation response, the value should be increased somewhat.

7. Simulation

7.1 Torque

We are getting closer to calculating the angular acceleration caused by a force on a rigid body. We have seen how to define the body's resistance to angular change with the inertia tensor, and we know the relationship between that inertia, the angular acceleration of the body, and the torque. So the only remaining issue is how to calculate the torque.

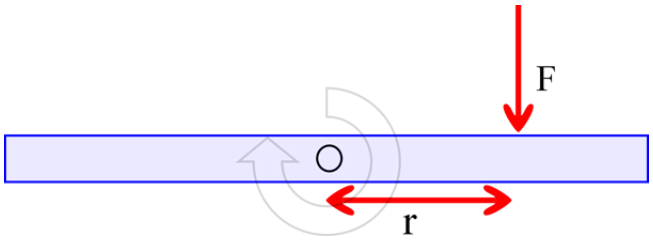

Figure 2. Calculating Torque - Relationship between linear and rotational force (i.e., torque).

You will recall that the torque τ produced by force F at distance r from the pivot point is calculated from:

$$\tau = dF \tag{14}$$

As we are simulating in three dimensions, the torque must be calculated for each of the three axes. This is achieved by taking the cross product of the distance vector and the force vector, as shown below in Equation 15:

$$\tau = r \times F \tag{15}$$

We have already calculated the force F acting on an object in the previous practical, by resolving all the active forces at each step of the simulation. The distance r is simply the vector between the object's centre of gravity and the position on the surface of the body where the force is applied. This position will become important in later practicals as we move on to collision detection and response.

7.2 Calculation of Acceleration

Now we have all the information required to calculate the angular acceleration of a body. The equation which we use is:

$$\tau = I\,\alpha \tag{16}$$

As τ and α are three dimensional vectors, and I is a three dimensional matrix, we must calculate the inverse of the inertia matrix $\Im^{-1}$, and calculate the angular acceleration from:

$$\alpha = I^{-1}\,\tau \tag{17}$$

Calculating the inverse of a matrix can be computationally expensive. For 'diagonal' matrices; however, it is very straightforward as each diagonal element is simply replaced by its reciprocal, while the non-diagonal elements remain zero. This means that it is straightforward to calculate the inverse inertia matrix for symmetrical objects. For non-symmetrical objects, a full matrix inverse calculation must be performed. There are functions readily available to do this. It should also be noted that, as the shape of rigid bodies does not change, the inverse inertia matrices can be calculated at load time (or in a pre-load step in the tool-chain), so that they do not need to be calculated every iteration of the simulation.

7.3 Numerical Integration

We have discussed how to calculate the angular acceleration of a rigid body caused by forces acting on that body. The next step is to translate that acceleration into an angular velocity and orientation, in order to spin and rotate our objects in the virtual world. This is achieved through the same numerical integration methods as for linear motion. The basic equation, at the heart of the numerical integration of angular velocity is:

$$\theta_{n+1} = \theta_n + \omega_n\,\Delta t \tag{18}$$

which is directly analogous to the linear calculation, and the implementations of Explicit and Implicit Euler Integration, Symplectic Euler Integration and Verlet Integration follow from that in exactly the same way as for linear motion. So the equations used by Semi-Implicit Euler Integration (or Symplectic Euler Integration) to calculate the orientation of a body from its angular acceleration for each frame of the simulation are:

$$\begin{aligned} \omega_{n+1} &= \omega_n + \alpha_n\,\Delta t \\ \theta_{n+1} &= \theta_n + \omega_{n+1}\,\Delta t \end{aligned} \tag{19}$$

In C++ these equations are encoded as follows (note that the order of these two lines of code is very important, as the velocity must be calculated before it is used to calculate the orientation):

```
1 // Semi-Implicit Euler
2 NextAngularVelocity = ThisAngularVelocity + ↩
      ThisAngularAcceleration * dt;
3 NextAngularPosition = ThisAngularPosition + ↩
      NextAngularVelocity * dt;
```

8. Implementation

The aim of this practical session is to expand your physics simulation understanding to account for rotational behaviour. You will construct a set of algorithms which calculate the torque acting on a body, calculate the angular acceleration caused by that torque, and then iteratively solve the differential equations governing the rotation of the body, in order to calculate its orientation and angular velocity at each frame of the simulation. To demonstrate that the physics update is working, you will add spinning to your rigid body objects.

Steps The steps of the algorithm which you will implement are:

1. Calculate the inertia matrix for our physics shapes, and the inverse inertia matrix, before starting the simulation
2. For each step of the simulation, calculate the torque acting on each object
3. Multiply the inverse inertia matrix by the torque to calculate the angular acceleration
4. Use Symplectic Euler Integration to calculate the angular velocity and the orientation from the angular acceleration

9. Summary

We have discussed how torques are used to calculate the rotation of bodies. We have discussed how to expand our numerical integration scheme, and how to use it to calculate the angular velocity and orientation of an object from its angular acceleration. The concept of moment of inertia was introduced and we added inertia tensors to our simulation, allowing objects to resist rotational forces. This concludes the work on moving objects around in our simulated world.

Acknowledgements

We would like to thank all the reviewers for taking time out of their busy schedules to provide valuable and constructive feedback to make this article more concise, informative, and correct. However, we would be pleased to hear your views on the following:

- Is the article clear to follow?
- Are the examples and tasks achievable?
- Do you understand the objects?
- Did we missed anything?
- Any surprises?

The practicals provide a basic introduction for getting started with cloth effects. So if you can provide any advice, tips, or hints during from your own exploration of simulation development, that you think would be indispensable for a student's learning and understanding, please don't hesitate to contact us so that we can make amendments and incorporate them into future practicals.

Recommended Reading

Code Complete: A Practical Handbook of Software Construction, Steve McConnell, ISBN: 978-0735619678

Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin, ISBN: 978-0132350884

Game Inverse Kinematics: A Practical Introduction (2nd Edition) Kenwright. ISBN: 979-8670628204

Kinematics and Dynamics Paperback. Kenwright. ISBN: 978-1539595496

Game Collision Detection: A Practical Introduction (Paperback). Kenwright. ISBN: 978-1511964104

Game C++ Programming: A Practical Introduction (Paperback). Kenwright. ISBN: 978-1516838165

Computational Game Dynamics: Principles and Practice (Paperback). Kenwright. ISBN: 978-1501018398

Game Physics: A Practical Introduction (Paperback). Kenwright. ISBN: 978-1471033971

Game Animation Techniques: A Practical Introduction (Paperback). Kenwright. ISBN: 978-1523210688