Workshop Series: Newtonian Mechanics

Benjamin Kenwright[1]*

Abstract
Newtonian mechanics rely upon Newton's laws of motion concerning the relations between forces acting and motions occurring. We explain and review Newtonian mechanics in terms of their relevance to real-time physics-based simulations, including, the basic structure of a physics simulator and the fundamental elements of unconstrained motion.

Keywords
Newton's Laws, Unconstrained Motion, Momentum, Contacts, Penetration Resolution, Collision Response, Impulses, Rigid-Body, Real-Time, Video Games, Interactive, Classical Mechanics, C++

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

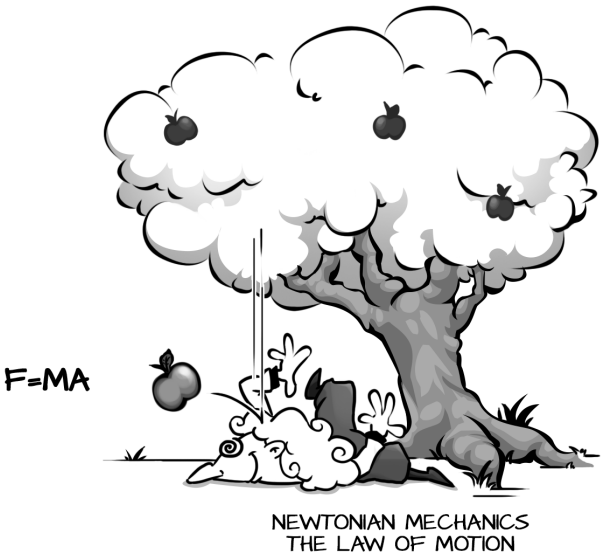

Figure 1. Sir Isaac Newton - Newton was born in a modest manor house in 1642 where he made many of his most important discoveries about light and gravity. Newton himself often told the story of how he was inspired to formulate his theory of gravitation by watching an apple fall from a tree.

Contents

Introduction

Physics The topic of this set of practicals is simulating the physical behaviour of objects through the development of various physics systems. You should possess the understanding of how to draw objects in complex scenes on the screen. We show you how to make those objects move and interact based on physical properties (e.g., mass and velocity).

Essentials The topic of this set of practicals is simulating the physical behaviour of objects in virtual environments, such as games, through the development of different physics simulations. The focus of this practical series is making the scene move and interact according to the laws of classical mechanics. Broadly speaking a physics simulator provides three aspects of functionality:

- Moving items around according to a set of physical rules
- Checking for collisions (or constraints) between items
- Reacting to collisions (or constraint violations) between items

The use of the word item, rather than object this is intended to suggest that physics can be applied to any and all elements of a scene - e.g., objects, characters, terrain, particles, or any part thereof. An item could be a crate in a warehouse, the door of the warehouse, the limb of a jointed character, the axle of a racing car, or even a snow flake particle.

1. Overview

Principles and Concepts Remember that, at this point, a rigid body physics system is still dealing with all the simulated objects - the new physical state of the simulated objects is not drawn to the screen until after all the physics update is complete, so the player will not see any of the intersections between objects, if they are successfully resolved.

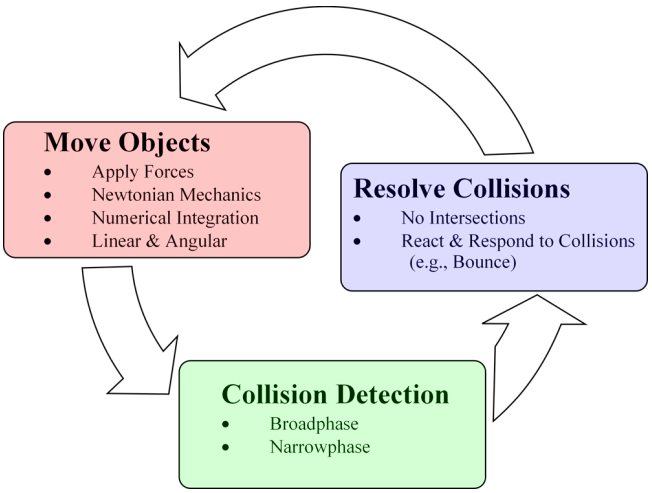

Figure 2. Systematic Approach - Uncomplicated Rigid Body Update Steps.

A practical physics system, e.g., those used in interactive environments, such as games, are based around Newtonian mechanics - i.e. the three simple rules of motion that you learned at school. These rules are used to construct differential equations describing the movement of our simulated items. The differential equations are then solved iteratively by the algorithms that are developed over this course. This results in believable movement and interaction of the scene elements.

The practicals then concentrate on preventing the various moving elements from intersecting with other elements in the scene. For example, a static scenes has no physical presence and any object can intersect with any other object. This is obviously not acceptable for a virtual world, such as games. In order to prevent solid objects from intersecting, we need to detect when they have intersected. In later practicals, we look at a suite of algorithms for detecting these intersections and resolving them. When an intersection is identified, the simulator needs to resolve the situation, by pushing the two intersecting items apart. This is achieved through the algorithms which simulate Newtonian mechanics, resulting in believable looking motion that bounces, rebounds and slides (i.e., friction).

This practical discusses the overall structure of a physics system within the context of real-time interactive scenes, such as games. Before getting into that, we should take a refresher course on the Newtonian mechanics upon which our physics systems will be based.

2. Newtonian Dynamics

A physics system for virtual scenes, such as games, are based around Newtonian dynamics, which are described by three fundamental laws of motion:

- A body will remain at rest or continue to move in a straight line at a constant speed unless acted upon by a force.
- The acceleration of a body is proportional to the resultant force acting on the body, and is in the same direction as the resultant force.
- For every action there is an equal and opposite reaction.

These laws will be familiar to you from school. We'll now consider each law in turn, and discuss their relevance in developing your physics simulator.

2.1 Newton's First Law

Often referred to as the Law of Inertia, this law states that an object will only change its velocity if there is a force acting on it. Consider a spacecraft in the vacuum of space - it will remain stationary until its boosters are started. The booster applies a forward force causing the spacecraft to move forward. If the booster is then stopped, the spacecraft will continue to move with constant velocity until a further force is applied (e.g. a steering force, a slowing force, or a collision).

This is a fundamental aspect of physics systems - all elements of the virtual scene which are controlled by the physics system are moved through the use of forces. If no force is being applied to an object, then its velocity does not change. Remember, this means that it will either remain stationary, or continue to move at a constant velocity (i.e. the same direction and speed). Of course, in the real world, a moving object in the room you are currently in will gradually slow down, due to friction from the surface it is moving along, or from the air it is moving through. For this behaviour to occur in our virtual world, these forces of friction must be simulated through the physics simulator.

2.2 Newton's Second Law

The second law of motion describes the relationship between the force on a body, and the resultant acceleration of that body. It is expressed mathematically as:

$$F = ma \tag{1}$$

where F is the force on the body, m is the mass of the body, and a is the resulting acceleration. Clearly the acceleration is proportional to the force, as stated in the second law, and the proportional factor is equal to the mass of the body.

The first law of motion tells us that when a force is applied to a body, it changes the object's speed - the second law defines that change. This is the equation which is at the core of a physics simulator - the physical movement of the objects in the virtual world is calculated by solving this equation for every frame of the system. We will see how that is achieved soon.

2.3 Newton's Third Law

The third law states that every action has an equal and opposite reaction. This means that whenever two physical objects interact, there is an effect on both of them. An obvious example is a collision of two pool balls - when one ball strikes another, the speed and direction of both balls is changed. Another oft-quoted example is of a weight on a table - there is a force downward from the weight (due to gravity) and there is an equal force upward from the table, keeping the weight on the surface. An alternative wording of the law is "the forces of two bodies on each other are always equal and are directed in opposite directions". Note that this wording states that the forces are equal, not the acceleration - the acceleration is inversely proportional to the mass of the object (as described in the second law), so a larger body will accelerate less than a smaller body. For example, bouncing an orange off a car causes a big change in the orange's velocity, but an imperceptible change in the car's velocity - although the force acting on both is equal.

Whereas the first two laws give us the basis for moving objects around in our physics simulator, the third law adds an element of realism and believability to how they interact. Taking account of the third law leads us to the collision detection and collision response routines that we will address in later practicals of this module.

3. Conservation of Momentum

There is one further physics law which we need to remind ourselves of - the law of conservation of momentum. This states that, if no external force acts on a closed system of objects, the momentum of the closed system remains constant. Momentum is the product of a body's mass and velocity. The total momentum of a system is the sum of the momentums of each object in that system. This is best illustrated with an example, see Figure 3 below:

Consider a body of mass m_1 travelling with velocity v_1^-, when it collides with a mass of m_2 travelling with velocity v_2^-. The respective velocities of the two bodies after the collision are v_1^+ and v_2^+. The following equation must hold true:

$$m_1 v_1^- + m_2 v_2^- = m_1 v_1^+ + m_2 v_2^+ \qquad (2)$$

The equation sums the momentum of the two objects before and after the collision, and equates them. Note, that all velocities must be expressed in the same context, so in this example, v_2^- and v_1^+ are negative numbers. Also note the nomenclature of adding a sign for values before the collision, and a + sign for values afterwards – we will use this naming convention extensively when we look at collision response in more detail An alternative way to state this law is that the centre of mass of any system of objects will always continue with the same velocity unless acted on by a force from outside the system. If we think of the two objects in the example as a system, then the velocity of their combined centre of mass can not change after the collision.

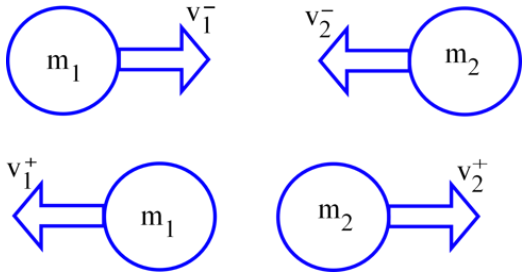

Figure 3. Conservation of Momentum - The law of conservation of momentum. ($^-$ indicates before the collision and $^+$ after the collision).

As with Newton's third law, the law of conservation of momentum will allow us to increase the realism introduced by our physics engine as we introduce collision response algorithms in later practicals.

4. Resolving Multiple Forces

We also require a quick reminder of how to resolve multiple forces acting on a single body. The body in the left half of Figure 4 below, has two forces acting upon it: a propulsion force F and a downward gravitational force G. The right hand figure shows how the two forces are combined to give the resultant force R:

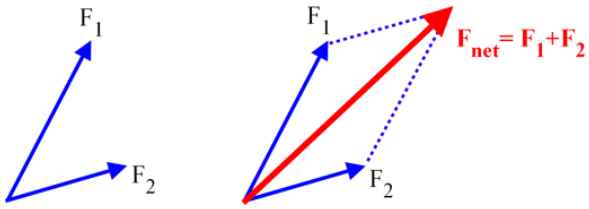

Figure 4. Summing Vector Forces - Vector Addition.

As you can see, the resolved force is simply the sum of all the forces acting on the body. Bare in mind that we represent forces with three dimensional vectors, comprising a x, y, and z component, so adding two vectors simply involves adding the relevant components of the vectors.

$$(F_{net_x}, F_{net_y}, F_{net_z}) = (F_{1_x} + F_{2_x}, F_{1_y} + F_{2_y}, F_{1_z} + F_{2_z}) \qquad (3)$$

or, more generally:

$$\sum F = (\sum x, \sum y, \sum z) \qquad (4)$$

The scalar size of the force is thus the length of the summed vector, and the direction of the force is found by normalising the summed vector. Remember, that the scalar size of a vector is found by summing the squares of the components and taking the square root. Normalising the vector then involves taking each component of the original vector and dividing by the length. Also bare in mind that square roots and divisions are expensive computationally, so only calculate the scalar size, and normalised vector, if they are required - a lot of your algorithms will only require the three components of the summed vector, which are much cheaper to compute.

5. Physics Engine Structure

Throughout this practical series the piece of software which provides the physical simulation is referred to as a Physics Simulator. Strictly speaking a simulator is a device designed to convert energy into useful mechanical motion. Within the context of a system, the physics simulator provides the motion of all of the elements of the virtual world; as such, it needs to be able to move any item in the world. At the risk of stretching the analogy, imagine designing a real system which needs to be able to move many different things in the real world - such a system would need a generic set of connections, and those connections would need to be built into everything it was expected to power.

Hence, the physics engine for a game tends to be a separate entity which links to the rest of the code through an interface; it doesn't care what the entities are that it is moving, it just cares about their physical properties, for example, size, weight, and velocity. This leads to the concept of a "physics engine" as a module which is distinct to the graphics, system, and audio code. Ideally a physics engine should be able to provide physical motion and interaction for any virtual scene, through a standard interface and data structure.

There are a number of commercially available physics system used in the games industry, such as PhysX, Bullet, and Havok - you will have noticed these logos during the start-up sequence of many games in recent years. Also, some of the larger publishers use an internally developed physics engine across multiple projects and studios. As the aim of this lecture series is to provide an understanding of how and why physics functionality works in games, we will not be using a physics engine developed elsewhere; instead you will be developing a physics engine of your own.

5.1 Update Loop

A physics engine typically operates by looping through every object in the scene, updating the physical properties of each object (e.g., position and velocity), checking for collisions, and reacting to any collisions that are detected.

The simulator loop, which calls the physics update for each object of interest, is kept as separate as possible from the rendering loop, which draws the graphical representation of the objects. Indeed, the main loop of the system can essentially consist of just two lines of code: one calling the render update, and the other calling the simulation update. The simulation should be able to run without rendering anything to screen, if the rendering and simulation aspects of the code are correctly decoupled. The part of the code which accesses the physics engine will typically run at a higher frame-rate than the rest of the game, especially the renderer. The physical accuracy of the simulation improves as the speed of the simulation is increased - games often update the physics at a rate of 120fps, even though the renderer may only be running at 30fps. As we will see in later practicals in this series, the physics engine works by iteratively solving differential equations representing the motion of the simulated objects, so the more frequent the iterations, the more accurate the results will be, hence the higher update rate of the physics engine.

Clearly, if there are large numbers of game entities requiring physical simulation, this can become a computationally expensive situation. In order to reduce the number of objects simulated by the physics engine at any time, similar techniques to those used in the graphics code for limiting the number of objects submitted to the renderer are employed. However, culling objects based on the viewing frustum is not a good way of deciding which objects receive a physics update - if objects' physical properties cease to be updated as soon as they are off-camera, then no moving objects would enter the scene, and any objects leaving the scene would just pile up immediately outside the viewing frustum, which would look terribly messy when the camera pans around.

A more satisfactory approach is to update any objects which may have an interaction with the player, or which the player is likely to see on screen, either now or in the near future. This is usually achieved by splitting the virtual world into a series of regions in some way - the simulation loop then only updates the objects which are in the regions currently of interest. The decision of which regions are currently of interest is made right at the start of the simulation loop, as it provides a very high level culling of candidate objects. It will typically involve selecting the region(s) where the camera and player are currently residing, and any regions which the camera or player may move into in the near future. As you can tell from this description, there is no single solution to this issue, and the algorithms will be crafted toward the specific simulation effect. For example, an open world virtual environment will be made up of many regions, and the algorithm deciding which ones are currently active will be based on numerous parameters including the current speed of the player, the activity of particularly relevant AI. Whereas a two-dimensional scrolling platform simulation is likely to consist of a chain of regions along the two-dimensional route, only updating the region where the player happens to be, and the next one along the chain.

When simulating the physical behaviour of many objects, it is easy to fall into the trap of updating some of the objects more than once in a single step of the simulation. For example, if body A collides with body B, but then body C collides with

body A, it is tempting to go back and update body A again with this new information. This approach can quickly lead to discrepancies due to some objects receiving far more updates than others, or even to an extended or infinite delay as the sequence goes round and round some inter-related objects. Care should be taken to only apply a physics update once for each object in each step of the simulation - any remaining intersections will be addressed in the following set of updates.

6. Physical Representation

Each item simulated by the physics engine requires some data representing the physical state of the item. This data consists of parameters which describe the item's position, orientation and movement. Depending on the complexity and accuracy of the physical simulation required, the data and the way in which it is used become more detailed. As ever, the simpler the physical representation, the cheaper the computational cost and, therefore, the greater the number of items which can be simulated. The three main types of simulation are particles, rigid bodies and soft bodies.

6.1 Particles

The simplest representation of physical properties is achieved through the particles method. In this case, items are assumed by the physics engine to consist of a single point in space (i.e. a particle). The particle can move in space (i.e. it has velocity), but it does not rotate, nor does it have any volume. This approach can be used for real-time particles, and also for other virtual items at times when speed of calculation is more important than accuracy - for example, when larger objects are sufficiently distant, or even off-camera, so that the player is unlikely to see intersections or lack of rotational detail.

6.2 Rigid Bodies

The most common physical representation system is that of rigid bodies. In this case, items are defined by the physics engine as consisting of a shape in space (e.g. a cube or a collection of spheres). The rigid body can move in space, and can rotate in space - so it has both linear and angular velocity. It also has volume - this volume is represented by a fixed shape which does not change over time, hence the term "rigid body". This approach is taken for practically everything in games where reasonable accuracy (or better) is required. More complex items, such as a spaceship, a tea-pot, or a dinosaur, are built up from a number of interconnected rigid bodies - each element of the skeletons and scene graphs is likely to be represented in the physics engine by a rigid body.

6.3 Soft Bodies

Items which need to change shape are often represented in the physics engine as soft bodies. A soft body simulates all the aspects of a rigid body (linear and angular velocity, as well as volume), but with the additional feature of a changeable shape - i.e. deformation. This approach is used for items such as clothing, hair, wobbly alien jelly-fish, an so forth. It is considerably more expensive, both computationally and memory-wise, than the rigid body representation.

It should be noted that a fully implemented physics engine will include options to simulate items at any of these levels of complexity at any time; it is up to the user to decide which items are simulated by which methods, appropriate to the scene and frame-rate constraints. Later practicals in the series consider these simulation approaches in a lot more detail.

7. Physics Shapes are not the same as Graphics Shapes

We have already discussed how the physics simulation should be as decoupled as possible from the rendering loop. This also applies to the data structures, and to the shapes and meshes of the game objects. Whereas the object which is rendered onto the screen can be any shape, made up of many polygons, it is not practical to simulate large numbers of complexly shaped objects in the physics engine.

Almost every object that is simulated by the physics engine will be represented as a simple convex shape, such as a sphere or cuboid, or as a collection of such shapes. As we'll see in later practicals, calculating collisions and penetrations between objects can be a very expensive process, so simplifying the shapes which represent the simulated objects greatly improves the required computation.

Each game object data structure contains a link to the graphical representation (e.g., the vertex lists and textures), that we dealt with in graphics), and the physical representation (where the object is, how fast it is travelling, the shape and size of its physical presence). So the data structure for an object such as a crate contains a link to the vertex lists which give the renderer detailed instructions on how to draw it, and a much more simple set of data defining the height, width and length of the cuboid which contains all vertices of the crate. Similarly a Christmas bauble data structure contains some very basic information on the size of the sphere which is used to simulate it in the physics engine.

8. Three Dimensions and Vectors

The physics engine which is developed in this practicals series is three dimensional - that is, it simulates the behaviour of objects within a three-dimensional environment, so all objects are modelled and simulated in the x, y and z axes. The concepts presented are just as applicable to a two-dimensional simulation, such as a 2D platform game or old-style space shooter, in which case only the x and y axes would be used.

It is also important to remember that the physical properties of the simulated objects are represented by vectors, rather than scalar parameters. This means that, not only is the position of an object in world space represented by a three dimensional vector (P_x, P_y, P_z), but the velocity and acceleration are also represented by three dimensional vectors. For example a ball falling vertically to the ground may have a velocity of (0, -1, 0), showing that the y component of the

velocity is negative while the x and z components are zero. A hover-ship which is moving in a vertical circle at constant speed will have sinusoidally changing values of velocity in the x and y components of the velocity vector.

Remember that speed is a 'scalar' quantity, whereas velocity is a 'vector', so the length of the velocity vector is equal to the speed.

$$S = \sqrt{V_x^2 + V_y^2 + V_z^2} \tag{5}$$

9. Implementation

During the course of this practical series you will create a different physics simulations. The implementation sections of the practicals include example code, and explanation, of how to achieve this. While graphics introduces new API commands and OpenGL functions to stay upto date with current system technology, this is not the case with the Physics-Based Animation practicals. You will be using the C++ that you already know to build your physics simulations. Consequently the example code should be seen as just that: an example of how to implement the theories and algorithms presented in each practical.

The aim of this practical session is to structure your framework program so that the physics engine can be implemented over the remainder of the practical series. You will separate the simulation update from the graphics update, introduce the basic data structures required by the physics engine, and use this framework to move an object around the environment independently of the graphics update.

You will implement the simulated objects as rigid bodies, so a 'Rigid Body' class is required. The code snippet below shows the member data that the rigid body class needs. The purpose of each variable is probably clear from their names, and we will gradually introduce the way in which they are calculated and utilised over the next couple of practicals. Each of your objects which is to be simulated by your physics engine requires this data, which should be separate from the data related to how the object is rendered (e.g. vertex lists and texture details).

Listing 1. Rigid Body Elements (note: for an uncomplicated particle system, we can exclude the rotational terms).

```
1  class RigidBody
2  {
3   // <------LINEAR ---------->
4   Vector3   m_position ;
5   Vector3   m_linearVelocity ;
6   Vector3   m_force ;
7   float     m_invMass ;
8
9   // <------ANGULAR --------->
10  Matrix    m_orientation ;
11  Vector3   m_angularVelocity ;
12  Vector3   m_torque ;
13  Matrix    m_invInertia ;
14 };
```

10. Summary

We have introduced the basic physics-based concept of motion, and had a quick refresher course of the physics required to develop the algorithms. The physics simulators which we will develop will be based on forces, so we have seen how to resolve multiple forces affecting a single body, and discussed the mathematical relationship between force and acceleration. In the next practical, we will discuss how to implement that relationship and how to translate an acceleration into a moving object in a virtual environment. Your module work should now be structured in such a way that the physics simulation is separate from the renderer update, and so that the physics implementation which you will develop can easily interface with the rest of your system (i.e., modular component programming).

11. Exercises

This practical only gives a brief taste of physics-based principles. As an exercise for the student to help enhance their understanding:

Intermediate

- Implement basic particle objects (e.g., spheres) and have them float around in a scene
- Implement a large scene with a vast number of bodies (e.g., >100)
- When a key is pressed apply random forces to the bodies to trigger motion
- Reversing the velocity of each object when it goes outside a pre-defined bounds (e.g., a box or sphere) - simple collision response effect

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