Physics Tutorial 1: Introduction to Newtonian Dynamics

Summary

The concept of the Physics Engine is presented. Linear Newtonian Dynamics are reviewed, in terms of their relevance to real time game simulation. Different types of physical objects are introduced and contrasted. Rotational dynamics are presented, and analogues drawn with the linear case. Importance of environment scaling is introduced.

New Concepts

Linear Newtonian Dynamics, Angular Motion, Particle Physics, Rigid Bodies, Soft Bodies, Scaling

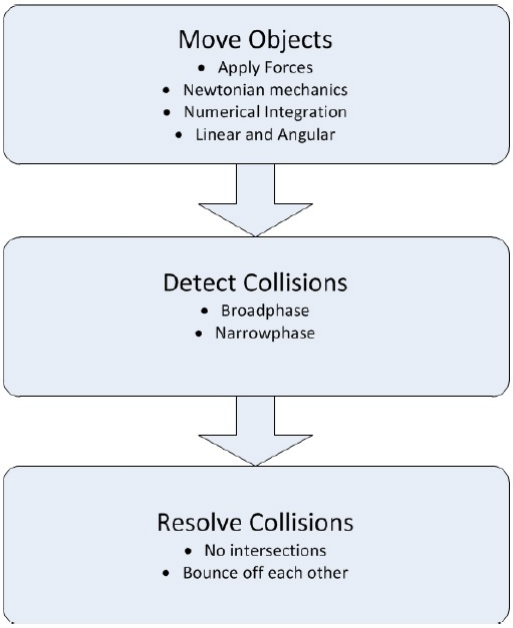
Introduction

The topic of this set of tutorials is simulating the physical behaviour of objects in games through the development of a physics engine. The previous module has taught you how to draw objects in complex scenes on the screen, and now we shift our focus to moving those objects around and enabling them to interact.

Broadly speaking a physics engine provides three aspects of functionality:

- Moving items around according to a set of physical rules
- Checking for collisions between items
- Reacting to collisions between items

Note 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 game scene – objects, characters, terrain, particles, etc., or any part thereof. An item could be a crate in a warehouse, the floor of the warehouse, the limb of a jointed character, the axle of a racing car, a snowflake particle, etc.



A physics engine for games is based around Newtonian mechanics which can be summarised as three simple rules of motion that you may have learned at school. These rules are used to construct differential equations describing the behaviour of our simulated items. The differential equations are then solved iteratively by the algorithms that you will explore over the course of this module. This results in believable movement and interaction of the scene elements.

Later tutorials concentrate on preventing the various moving elements from intersecting with other elements of the scene. So far, the scenes which you have developed have had no physical presence to them any object can intersect with any other. This is obviously not acceptable for a game world. In order to prevent solid objects from intersecting, we first need to detect when they have intersected. Later tutorials address a suite of algorithms, and two distinct approaches, for detecting these intersections (or *collisions*). When a collision is identified, the engine needs to resolve the situation, by pushing the two intersecting items apart. This is achieved through the algorithms which simulate Newtonian mechanics mentioned above, resulting in believable looking bounces, rebounds and friction.

The physics engine itself is discussed in this tutorial with regards to its position within your overall game engine framework. The subsequent focus of this lesson is the physical foundations on which it is constructed, beginning with a refresher on Newtonian dynamics, before progressing onto more advanced topics.

The Physics Engine

Throughout this tutorial series the piece of software which provides the physical simulation is referred to as a Physics *Engine*. Strictly speaking an engine is a device designed to convert energy into useful mechanical motion. Within the context of a game, the physics engine provides the motion of all of the elements of the game 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 engine which needs to be able to move many different things in the real world - such an engine 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 size, weight, velocity, etc. This leads to the concept of an "engine" as a module which is distinct to the game code, renderer code, audio code, etc. Ideally a physics engine should be able to provide physical motion and interaction for *any* game, through a standard interface and data structure.

```
//A whole physics engine in 6 simple steps =D
//1. Broadphase Collision Detection (Fast and dirty)
//2. Narrowphase Collision Detection (Accurate but slow)
//3. Initialize Constraint Params (precompute elasticity/baumgarte
//  factor etc)
//4. Update Velocities
//5. Constraint Solver     Solve for velocity based on external
//  constraints
//6. Update Positions (with final 'real' velocities)
```

PhysicsEngine::UpdatePhysics

There are a number of commercially available physics engines 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 larger publishers use internally developed physics engines 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.

While we will not be using a physics engine developed elsewhere (you will be developing a physics engine of your own), the principles you become familiar with over the course of the module should give you a deep familiarity with the complexities involved in physics programming, and that knowledge maps well to an understanding of effective use of any commercial physics engine.

You are encouraged to explore commercial physics engines (many are available for hobbyist use), to gain familiarity with how industrial employers might handle function calls, etc., but you will not be assessed on this.

Update Loop

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

The simulation loop, which calls the physics engine 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 game 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 later in this series, the physics engine works by iteratively solving differential equations representing the motion of the simulated objects. The more frequent the iterations, the more accurate the results will be, hence the higher update rate of the physics engine when compared to the renderer.

Clearly, if there are large numbers of game entities requiring physical simulation, this becomes 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 game 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 game. For example, an open world game 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, etc. Whereas a two-dimensional scrolling platform game 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. Moreover, the subsequent resolutions for a single object can actually lead to a less consistent simulation.

For the remainder of this tutorial, we will focus on the science which underpins real-time physics simulation, rather than the engineering. There'll be plenty of both in subsequent lectures, but in order to grasp the application of more advanced, mathematical concepts, it's crucial to have a solid grounding in the basic, underlying physics.

Newtonian Dynamics

A physics engine for games is 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 may be familiar to you from school. We'll now consider each law in turn, and discuss their relevance to developing a physics engine for games.

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 a physics system for games - all elements of the game which are controlled by the physics engine 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 game world, these forces of friction must be simulated through the physics engine, often in an abstract way.

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 game physics engine - the physical movement of the objects in the game world is calculated by solving this equation for every frame of the game. We will see how that is achieved soon.

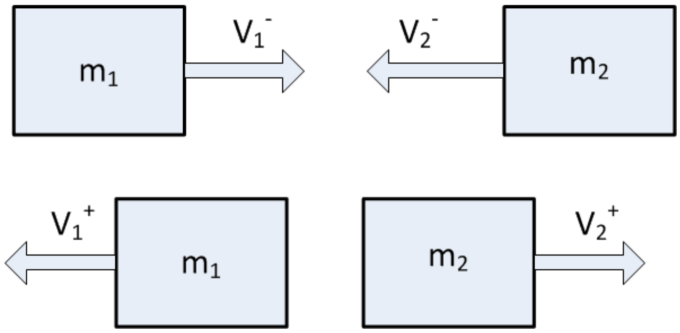
Newton's Third Law

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 engine, 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 tutorials of this module.

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 momenta of each object in that system. This is best illustrated with an example.



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:

$$m_1 v_1^- + m_2 v_2^- = m_1 v_1^+ + m_2 v_2^+ \tag{2}$$

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.

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 tutorials.

Three Dimensions and Vectors

The physics engine which is developed in this tutorial 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.

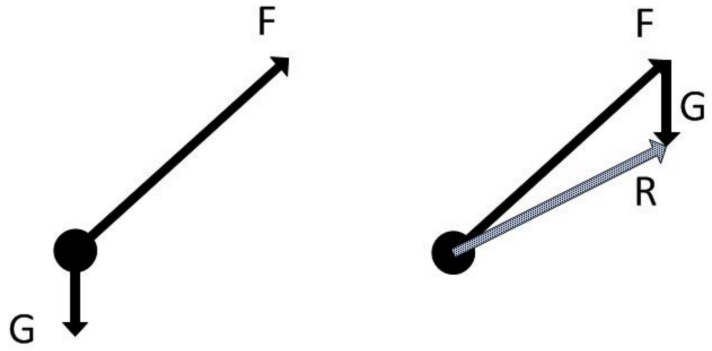
It is also important to remember that the physical properties of the simulated objects are represented by vectors, rather than scalar parameters. This means that, not only is the position of an object in world space represented by a three dimensional vector (P_x, P_y, P_z), but the velocity and acceleration are also represented by three dimensional vectors. For example a ball falling vertically to the ground may have a velocity of $(0.0, -1.0, 0.0)$, 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.

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

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 the figure 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



As you can see, the resolved force is simply the sum of all the forces acting on the body. Bear 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.

$$(R_x, R_y, R_z) = (F_x + G_x, F_y + G_y, F_z + G_z)$$

or, more generally

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

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 bear 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.

Defining Our Linear Motion Variables

Linear motion concerns itself with three physical properties of an object:

- a is the acceleration of the object. If an object's speed changes over time, it has acceleration (acceleration can easily have negative vector components in 3D space, while still having high absolute magnitude; as such, we avoid the term 'deceleration').
- v is the velocity of the object - how much it is displaced over time. If an object is moving, it has velocity.
- s is the displacement, or position, of the object.

Introducing Angular Motion

Before we move on to discuss the properties which underpin angular motion in a Newtonian physical simulation, we need to introduce (and reintroduce) a couple of key concepts.

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.

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

Consequently a full rotation of 360^o is expressed as 2π radians; half of a complete revolution (i.e. 180^o) is π radians and so on. In most game 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 follows:

```
1 if (ThisAngle > TWO_PI) ThisAngle -= TWO_PI;
2 if (ThisAngle < -TWO_PI) ThisAngle += TWO_PI;
```

Revolutions

Note that a defined constant is used (TWO_PI), instead of multiplying π by two for every object. Also 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.

Quaternions

Here's a little spanner thrown into the works. If we're dealing with objects which exist in space, those objects will have an orientation. Orientation is important for a multitude of reasons. In the simplest case, a sphere, orientation informs texture mapping. For more complex objects, orientation affects both collision detection and collision response. In short, knowing, tracking, and changing the orientation of an object is essential to it 'looking good' and 'behaving well'.

The orientation of an object, and how quickly that orientation changes, are modelled through angular mathematics. The concept is pretty much similar to that of linear motion, so we have direct counterparts 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 can be generalised to three dimensional parameters represented by vectors and quaternions.

In the case of Θ, this can be represented as a quaternion for orientation about an axis. ω is a vector representing angular velocity about the same axis and scaled by the magnitude of the rotation. Angular acceleration has a similar relationship with torque (discussed later) as linear acceleration does with Force (as discussed previously).

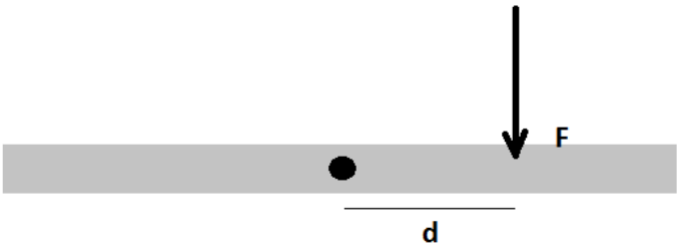
Quaternions were introduced during the skeletal animation portion of the graphics tutorials. In the context of our physics modelling, Θ can be defined:

$$\Theta = \left(x \sin\left(\frac{\theta}{2}\right), y \sin\left(\frac{\theta}{2}\right), z \sin\left(\frac{\theta}{2}\right), \cos\left(\frac{\theta}{2}\right) \right)$$

where $\hat{\mathbf{n}} = (x, y, z)$, and represents the axis of rotation, and θ represents the angle of that rotation.

Torque

Now we've covered some of the tools needed to understand how we'll move forward with our exploration of angular motion, we need to start framing the problem in a familiar way. *Torque* is the result of a force applied to an object, a given distance from its pivot point. As force is the fundamental building block of our physics simulation in a linear sense, so its angular analogue is here. Consider the figure below:



Torque τ produced by force F at distance d from the pivot point is calculated from:

$$\tau = dF$$

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.

$$\tau = d \times F$$

We have already calculated the force F acting on an object earlier in this tutorial, by resolving all the active forces at each step of the simulation. The distance d 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 tutorials as we move on to collision detection and response.

Defining Our Angular Motion Variables

Now we have an analogue for force in our angular motion, we should clarify the other analogues which exist, to make understanding what follows a little more straightforward. As with the linear variables, the actual interactions between these variables are explicitly explored in a future tutorial.

- θ is the angle of our computation - this is analogous to position or displacement in the linear case.
- ω is the angular velocity of our rotating object - if an object's angle is changing, it has angular velocity. This is analogous to v in the linear case.
- α is the angular acceleration. If ω is changing, then an object has angular acceleration.

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* 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:

$$\tau = I\alpha$$

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:

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

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 $I_{zy} = I_{yz}$.

Calculation of Acceleration

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

$$\tau = I\alpha$$

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

$$\alpha = I^{-1}\tau$$

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.

Symmetrical Objects

Many world elements in games can be simulated by the physics engine as completely symmetrical objects – i.e. objects where the weight is evenly distributed around the centre of gravity – even many objects which, in the real world, would not actually be symmetrical, such as barrels, bricks, crates, etc. 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:

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

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

$$\tau_x = I_{xx}\alpha_x$$

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.

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 engine 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.

Solid Sphere

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

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

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

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

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.

Solid Cuboid

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

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

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

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

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.

Asymmetrical Objects

In the real world, almost no objects are symmetrical. However in a game world, simulating non-symmetrical objects is more computationally expensive than 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 inertia 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:

$$I_{xx} = M\int_V (y^2 + z^2)\,dV \quad I_{yy} = M\int_V (x^2 + z^2)\,dV \quad I_{zz} = M\int_V (x^2 + y^2)\,dV$$

and the non-diagonal elements are:

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

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

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.

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 actual in-game particles, and also for other game 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.

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 that we saw during the graphics course is likely to be represented in the physics engine by a rigid body.

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 jellyfish, etc. It is considerably more expensive, both computationally and memory-wise, than the rigid body representation, so it is only used where it is specifically required and when the player will see the resultant behaviour.

	Velocity	Angular	Volume	Deformation
Particle	Y	N	N	N
Rigid Body	Y	Y	Y	N
Soft Body	Y	Y	Y	Y

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 game and frame-rate constraints. Later tutorials in the series consider these simulation approaches in a lot more detail.

Physics Shapes are not the same as Graphics Shapes

We have already suggested that physics simulation should be as decoupled as possible from the rendering loop, and facilitating this will be explored later on. This principle also applies, however, 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 complicatedly 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 other straightforward polyhedron, or as a collection of such shapes. As we'll see in later tutorials, 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. This is the case even when employing the more advanced collision and response techniques explored later in this lecture series.

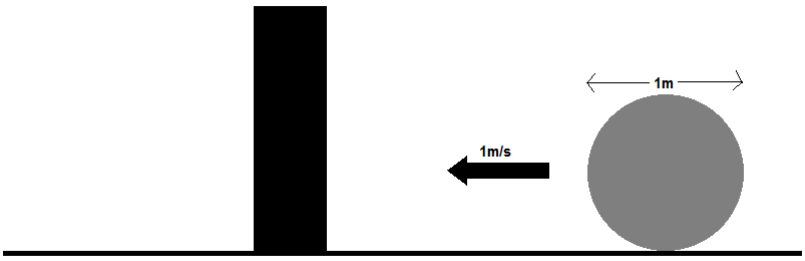
Each game object data structure contains a link to the graphical representation (the vertex lists, textures, etc. that we dealt with on the graphics course), and the physical representation (where the object is, how fast it is travelling, the shape and size of its physical presence, etc.). So the data 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.

On Environment Scaling

One crucial, and deceptively simple, concept to grasp when constructing a physics engine is the relationship between motion of entities and their size: the relative scaling of the environment. Many apparent glitches and aberrant behaviours demonstrated by a physics system can ultimately be traced back to an error in scaling.

Consider the example below. A sphere, 1m in diameter, is travelling at 1ms^{-1} towards the wall.



If we assume our fixed timestep is small, 1/120th of a second. In this case, an interface between the two bodies should be reliably detected and resolved. The sphere can only have travelled 1/120th of a metre between frames of our simulation, meaning its penetration of the wall between frames can only be of the order of 0.008m, or 0.8% of its diameter.

But what happens if our sphere is 2mm in diameter? In that scenario, there is a very real possibility that an interface between the sphere and the wall might be overlooked - the sphere might pass through the wall, because in one frame it is on one side, and in the next it has already traversed the space the wall occupies. Further to that, even if a collision is detected, the normal of that collision (the direction of approach of the sphere, essentially) might be inverted (i.e., the physics engine might deduce that the collision is happening from the other direction, because the sphere interface is detected after it has travelled over halfway through the wall).

Another common issue is the reverse problem. Imagine you have a sphere 200m in diameter, travelling at 0.02ms^{-1}. It is easy to conclude, during debugging, that the sphere isn't moving at all - particularly without a graphical frame of reference from the environment (such as a skybox). The problem is further complicated if using meshes of vertices to define a complex object - you need to ensure that the mesh coordinates are scaled to match the physical model of the entity, and that the physical model of the entity is also scaled sensibly for the environment.

For this reason, physics engines tend to employ a common scaling system to define the size of objects and the magnitude of velocities. When you set a cube to be 1.0 units wide, and a sphere to be 1.0 units wide, they should appear to be the same size in your environment, irrespective of the vertex values defining the cube. Similarly, if you apply a velocity of magnitude 1.0 units per second to a cube 2.0 units wide, it should traverse 1.0 unit every second, not 1.0 times its own width.

Implementation

During the course of this tutorial series you will create a physics engine. The implementation sections of the tutorials include example code, and explanation, of how to achieve this. While the series on *Programming for Games* and *Graphics for Games* introduced new C++ commands and OpenGL functions as they progressed, this is not the case with the Games Technology tutorials. You will be using the C++ that you already know to build your physics engine. Consequently the example code

should be seen as just that: an *example* of how to implement the theories and algorithms presented in each tutorial.

Another difference between this module and your previous modules is that practical work is generally associated with the entire day's learning objectives, rather than a single tutorial's. This is due to the double-lecture nature of some material, where theory needs addressing before implementation can be approached (in part, so as to understand where implementation has to diverge from theory).

Review the Practical Tasks hand-out for Week 1 of the module. Undertake the tasks suggested during the next practical session.

Tutorial Summary

We have introduced the concept of a physics engine, and had a quick refresher course of the physics required to develop the algorithms. The physics engine 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. We have extended these principles to the rotational dynamics of objects within our environment. In the next tutorial we will discuss how to implement these relationships and how to translate acceleration and torque into a moving, spinning object in a game environment.