

# Chapter 2

## Point Mass Dynamics

- **Newton Mechanics**
- **Gravity and Drag Forces**
- **Forces acting in Fluids**
- **Simulating Objects in Water**

*"The grass is always greener in the neighbor's yard.  
So let your dog shit all over it."  
(Marc Hebert)*

### 2.1 Introduction to Math and Physics

The first part of this chapter will introduce you to math and physics foundations. Here you will learn everything you need to know to understand what is going on in the remainder of this chapter. Since we don't have much time to waste I recommend you to fasten your seat belts and switch to your *math-fast-forward* mode. If you want to dive deeper into physics and math in general I would recommend you just reading a good physics textbook. For all German tongues among you [Paus02] is an excellent read. It covers a hundred of times more ground than what you will need here but its part on mechanics is very good, straight forward, and easy to understand.

#### 2.1.1 Rate of Change aka Derivatives

Before we can start diving into topics of game dynamics I want to repeat some basic math concept. The single most important concept you need to

know when dealing with physics is the notion of *rate of change*. From your basic calculus course you should remember that mathematicians define functions as shown below:

$$f(x) = a \quad (\text{read: } f \text{ of } x \text{ equals } a)$$

Where **a** is a placeholder for any kind of function you can think of – with a single unknown variable like **x** in this example. Now it is possible to derive another function from this one. The first derivative of the above function would be written like the following:

$$f'(x) = b$$

Again, the variable **b** in this equation is just a placeholder for the derived function **a**. Note that a lot of textbooks do not use the ' sign to indicate a derivative but follow Newton's notation and put a dot on top of the **f** instead.

So far so good. Don't worry, I won't show you how to build the derivatives of certain functions based on deriving rules. For the remainder of this book you only need to know what a derivative of a function is. Actually, the derivative of a functions has several interpretations such as describing the slope of the tangent lines of the curve representing **f(x)**. But the most important interpretation of the first derivative is the rate of change. If **f(x)** represents a quantity at any **x** the then the first derivative **f'(x)** represents the instantaneous rate of change of **f(a)** at **x=a**. Hence the first derivative is also written like this:

$$f'(x) = \frac{df}{dx} = \frac{dy}{dx} \quad (\text{read: derivative of } y \text{ with respect to } x)$$

In this equation **dy/dx** is called *differential coefficient* of **y** with respect to **x** and doesn't represent a fraction in such context. Its just one kind of notation that says **dx** is a very slight change in **x** and **dy** is a very slight change in **y** – roughly spoken. This notation simply reflects the fact that the first derivative of a function represents a ratio of change. Now, the derivative of a function is another function. That means you can also derive from the derivative, and so on. Note that the derivative of a derivative of

$f''(x)$  is the second derivative  $f''(x)$  of  $f(x)$ . Of course there are conditions that say if a derivative exists for certain values of  $x$  or if it exists at all. But luckily we are not interested in the calculus stuff. We want to learn about physics and game dynamics. Hence we don't need to know about those conditions and rules for building the derivative of a function.

## 2.1.2 Velocity as Rate of Change

Let's get back to the topic at hand. The most intuitive and well know example for the rate of change is the velocity. The velocity of an object represents the change in position of the object:

$$v = \frac{\Delta x}{\Delta t}$$

In math or physics equations the Greek letter  $\Delta$  (delta) is used to represent an amount or a quantity and its value represents a change as opposed to a specific value at a certain time step. The above equation can be read as: Velocity equals change in position with respect to change in time. Hence it means that velocity is the amount of distance traveled over some period of time. Please note that the  $x$  in this equation has nothing to do with the variable  $x$  used in the function  $f$  in the paragraphs above.

Okay, so here's your first physics equation. Lets make some use of it and calculate the velocity of a car that drove a distance of 50 miles within one hour:

$$v = \frac{50\text{miles}}{1\text{hour}} = 50\text{mph}$$

That's pretty easy, isn't it? But it also reveals a problem of this equation. Lets assume the driver of the car started his little journey at his home and drove across the town and through some villages to visit his grandma. Of course he did not drive constantly at 50 mph. He needed to accelerate his car at first. Then he would also have to break every now and then to stop at a traffic light, for example. The bottom line is that this equation to

calculate the velocity of an object is only useful to calculate the average velocity of an object with respect to the time period  $\Delta t$ .

To compensate for this drawback mathematicians would say that the equation is perfectly valid for the real velocity of an object if you can match one condition: The value of  $\Delta t$  needs to go to 0. If the amount of time goes to 0 then the velocity this equation represents is not the average velocity but the instantaneous velocity. Now you know that it is not so easy to divide a term by 0. That is why mathematicians use the following notations to express this condition:

$$v = \lim_{\Delta t \rightarrow 0} \frac{\Delta x}{\Delta t}$$

In this equation **lim** means limes or threshold value and the variable that goes to a certain threshold as well as the threshold are written below the **lim** keyword. So this seems to be a dead end, right? Now you know that this equation represents the real velocity of an object as long as you keep the delta value of the variable **t** as small as possible. Does that ring a bell? You want  $\Delta t$  to represent a slight change in **t** which will also mean that  $\Delta x$  is only a slight change in **x**. So of course you can also write this equation like so:

$$v = \lim_{\Delta t \rightarrow 0} \frac{\Delta x}{\Delta t} = \frac{dx}{dt}$$

This tells you that the velocity is a derived function. In other words, if you have a function **f(t)** representing the position of an object with respect to time you can build its derived function **f'(t)** representing the function to calculate the real velocity of the object for each **t** as opposed to the average velocity.

Now we go one step further. The velocity is the rate of change in position of a certain object. But then the velocity can also change with respect to time. Just think about the example used above with the guy going to his grandma with his car. He will start with a speed of 0 mph, then floor his accelerator to go 50 mph, hit the breaks to stop at a traffic light and so on.

So there is a change in velocity which can be represented by deriving the function used for velocity:

$$v' = \lim_{\Delta t \rightarrow 0} \frac{\Delta v}{\Delta t} = \frac{dv}{dt}$$

Of course the rate of change of the speed is acceleration. Note that using arbitrary delta values for the velocity and the time you can only calculate the average acceleration so I already added the step of using the differential coefficient. Still, the acceleration is not constant and hence we are talking of the average acceleration at the specific time interval.

Well, now that you have a basic understanding of this calculus we can start discussing how this relates to programming game dynamics. The next section will show you how to use your new knowledge to add real physics to your games.

## 2.1.3 Calculating Objects' Movements

In the preceding section you learned that you can get an object's velocity if you know a function describing its position with respect to time. You can then calculate its acceleration based on the change in velocity. That's all nice, but of course this isn't the way we use these equations because normally we don't have the function representing an object's position. Actually, we do it the other way round since what we want is to find a function to calculate our object's position. Just take another look at the equation representing the average acceleration of an object:

$$a = \frac{\Delta v}{\Delta t}$$

By rearranging this equation you can calculate the average velocity of the object:

$$\Delta v = a \cdot \Delta t$$

The change in velocity is just the acceleration times the interval of time you are taking into consideration. With the change in velocity you can now modify the velocity of the object:

$$v_+ = \Delta v$$

If you continue with the reverse engineering and look at the original equation used for the velocity of an object you can proceed towards your goal of finding the change in position of the object:

$$v = \frac{\Delta x}{\Delta t}$$

You can rearrange this equation to represent the change in position of an object:

$$\Delta x = v \cdot \Delta t$$

The change in position of an object equals the velocity of the object times the elapsed time. And now you can combine those two equations into a single one representing the change in position of an object:

$$\Delta x = (v + a \cdot \Delta t) \cdot \Delta t$$

Using this equation you can add all kind of game dynamics to your 3D objects. Just think about what you want the physics or game dynamics to do for you. You want them to move your objects based on real world physics. And that's how you can do it. All you need to have is the starting position  $\mathbf{x}$  and the initial velocity of an object. You can then calculate the acceleration of the object as well as the elapsed time since the last frame. Now you can calculate the distance the object has moved since the last frame and finally modify the object's position by adding this distance:

$$x_+ = (v + a \cdot \Delta t) \cdot \Delta t$$

Before we can now take care of the last unknown variable in this equation let me tell you that this equation does not only hold true in one dimension.

It is also perfectly valid in two and three dimensions where the position, the velocity, and the acceleration are three-dimensional vectors:

$$\vec{X}_+ = (\vec{V} + \vec{A} \cdot \Delta t) \cdot \Delta t$$

That's great. Now you know that you don't need to know a function representing the position of an object with respect to time. And as long as you keep the value of  $\Delta t$  small enough you can accept that you are only using average values for the acceleration and velocity in the given time interval as the average will get closer to the real value at a specific time. But still there is a problem. How do you get the acceleration of an object in order to be able to calculate the velocity or the change in position at all?

You can read the answer to this question in the next section.

## 2.2 Its all about Sir Isaac Newton

For quite some time mankind had a lot of misconceptions for how our universe works. One of those fallacies was that all moving objects naturally come to a stop. Of course that is what our world looks like so it was a valid assumption. Just go to a chair next to you and push it. It will slide a bit but then it will eventually come to a stop. Next, try kicking a ball. It will roll for a way longer distance but then again, it will come to a stop.

In 1643 a guy named Isaac Newton was borne in Woolsthorpe, England. He was a very talented guy in calculus, mechanics, and several other fields. I think you all know the fairy tale about this Newton guy sitting under a tree and being hit by an apple falling down on his head. Besides this story the single most important thing about Newton is that he was the first one who wrote down detailed laws describing how objects move in our universe. Even if he was not necessarily the first one to discover those laws.

Newton's so-called first law of motion explains that moving objects do not naturally come to a stop. There is always a force required to slow down objects to a halt. In most cases such a force is friction between surfaces or

friction and drag in fluids like air. Newton's second law of motion then describes how to calculate the amount of a force:

$$F = m \cdot a$$

This equation is surprisingly simple by saying that a force equals the mass of an object times its acceleration. As you will see throughout this book this equation is a very mighty one used in a lot of calculations. Actually, we can already make a good use of this equation. We came to this paragraph because we were asking how to calculate the acceleration of an object so that we could calculate its change in position. Well, here's the answer. Just rearrange Newton's second law of motion:

$$a = \frac{F}{m}$$

Now you only need to know the total force acting on an object as well as the mass of the object. Then you can calculate its acceleration, which means that you can calculate its velocity as well. That in turn means that you can calculate the position of the object.

*There is also Newton's third law of motion that says that for each action there is an equal and opposite direction. This might sound strange at first, but it's the basic principle that let's rocket engines work for example. The engine generates a strong force acting through its nozzle. Therefore, the rocket itself experiences an equal force acting in the opposite direction and thus pushing it forwards.*

But now it seems that we still have the same problem. In most cases we will know the mass of an object. If you want to simulate a racing game you will know the mass of the cars involved. But how the hell do you know the force acting on the cars – or an object in general?

Here is the good news: In physics we believe that nature does only know four different forces. Namely they are the gravitational force, the electromagnetic force, the weak nuclear force, and the strong nuclear

force. That means all physical problems you can think of will have a solution if you only take those four forces into consideration.

And here is the bad news: In real life doing this is way too complicated. That is why physicians have those thick textbooks featuring dozens and dozens of equations. In practice physicians would do a lot of experiments to measure the amount of force existing in certain situations. Based on those experiments they would then come up with a more or less simple equation that lets you calculate this force. Of course those equations do only hold true in similar environments under similar conditions. But they are all we need to calculate the total force acting on an object.

To conclude, the only thing we are really interested in knowing is the change of position of an object. You can easily calculate this if you know the acceleration of the object. But to get the acceleration of the object you need to know the total force acting on the object. Hence, the remainder of this book will mostly deal with different equations that enable you to calculate the forces acting under certain conditions, such as an upward force due to buoyancy in fluids, for example.

## **2.3 Primary Forces influencing Objects**

In this paragraph I want to introduce you to the main forces that influence a so-called *point mass*. For each force you will see an equation how you can calculate the amount of the force that you need to apply to your object to add up to the object's total force. With this total force you can then calculate the object's acceleration.

### **2.3.1 Point Masses and Linear Dynamics**

Before we start this discussion, I should tell you what exactly a point mass is, right? Even from elementary school you will remember that mathematicians and physicians like to simplify problems to be able to calculate approximations for problems that are too complex to come up

with an exact solution. So before we move on to more or less advanced objects simulating all aspects of physics, I want to simplify things first.

I assume that you are interested in physics and game dynamics and that you already had a look or two at some tutorials, or even engines about game physics. Then you will have heard the name *rigid body* all over the place. Such a rigid body is an object that enables you to simulate real physics. A point mass is basically an object that has a mass but no volume. It is an infinitesimal small object, which we call a point.

In this chapter you will learn how to fire all kind of forces onto your point mass and simulate the physics that will then move the point mass. So where is the simplification? Due to the non-existence of a volume for a point mass there is no need to take care of the orientation of the point mass. In other words, this means point masses, unlike rigid bodies, do not rotate. In mechanics you can split the treatment of an object into two separate parts. On the one hand you have the linear dynamics, which is what we will discuss in this chapter. On the other hand there are the rotational or angular dynamics, which we will omit for the moment since they required the object to have a volume.

Later in this book when we discuss rigid bodies, you will see that taking care of the rotation of an object due to forces acting on it is a complex task. That is why nearly all textbooks about game physics will usually start discussing point masses. Once you get a grip on point masses you will have a good head start into physics journey. You can then concentrate on the complex issues involved with angular dynamics.

## **2.3.2 Gravity, or why Apples fall on Heads**

The first force most of you would name is gravity. The gravitational force is what keeps our feet on the ground and what makes things start falling down. Interestingly we still can't tell why gravity is there or how it works. Well, from the General Relativity's point of view you can say that gravity is a consequence of the fact that the space-time isn't flat, but warped by it's mass and energy that it contains. So for example, the earth doesn't move

on an orbit around the sun because of gravity, but it's following the straightest path as part of a curved space. But again, this is just theory and as of now nobody knows if this is really how gravity works. We can only say what happens if a gravitational force is present and we can even predict where gravitational forces occur. The bottom line is that each object that has a mass does create a gravitational force, which attracts all other objects in its influence radius towards its center of mass.

Normally, this gravitational force is too small to be noticed. But if you look at big objects like the earth you will find that there is a noticeable force indeed. So all objects inside earth's gravity field are pulled towards earth's center point. This effect decreases with distance from the center point and you can already measure a difference in gravity at sea level and on top of a high mountain.

Since the amount of gravity depends on the mass of the object causing the gravitational force smaller objects such as the moon have less gravity on their surface. Hence they let astronauts jump higher and further than it is possible on earth's surface. On the other hand bigger objects such as the sun have a more powerful gravity field and can keep a whole bunch of planets in line over a distance of 3 billion miles and more.

But after all we are only interested in the force **F<sub>g</sub>** that acts on an object inside the gravity field. That leads us back to Newton and his second law of motion:

$$F_g = m \cdot a$$

Note that this time we are not dealing with the total force acting on an object and its total acceleration. Now, we are only interested in the force due to a gravity field. The mass of the object remains unchanged. The value of acceleration is now the acceleration caused by the gravity field. As you know a gravity field pulls objects towards a certain point. It does so by applying an acceleration to the objects.

If you know the gravity field and the object that is creating the gravity field then you can calculate the acceleration that acts on objects for each point inside the gravity field. I don't want to distract your attention now by

explaining how to calculate the exact acceleration, so please refer to your favorite mechanics website or web search engine. You won't believe the number of sources you can find online that will explain you all kind of math or physics stuff.

From your first physics class you will still remember that the acceleration in earth's gravity field is about  $9.81 \text{ m / s}^2$  close to sea level. We can take this value as constant for all kinds of simulations inside the lower earth's atmosphere since it will only slightly change in different altitudes.

So, finally, here is your first equation to calculate the amount of one force that is acting on your objects:

$$F_g = m \cdot 9.81 \quad [ \text{kg} * \text{m} / \text{s}^2 = \text{N} ]$$

Again, this equation is also valid in three dimensions. The acceleration due to gravity will become a vector  $\mathbf{a}( 0.0, -9.81, 0.0 )$  that has a downwards component only. The corresponding force, which is measured in Newton, will be a vector, too:

$$F_g(m \cdot 0.0, -m \cdot 9.81, m \cdot 0.0)$$

Okay, that one is pretty easy. As long as you know the amount of acceleration that acts in a gravity field at the position where your object is located you can calculate the force pulling your object. The neat thing is that you are of course not restricted to have earth's gravity acting downwards. In a weird land, odd things happen. If you want the gravity to double and to act upwards you can just change the corresponding acceleration vector  $\mathbf{a}( 0.0, +20.0, 0.0 )$ . The rest of your code remains untouched but your simulated world will look totally different.

If gravity is the only force acting on your object then you could of course directly use the gravity acceleration vector to calculate your object's velocity. But normally there will be some more forces acting on your body at the same time. Hence you need to calculate the gravitational force and add it to the amount of total force acting on your object. So let's move on to the next force that will normally influence your objects.

### 2.3.3 Drag and Friction, or why things are slowing down

As I've previously mentioned, mankind did believe for quite some time that all moving objects naturally come to a stop. This is a correct observation in earth's atmosphere but it's a wrong assumption to generalize this kind of behavior. Actually, there are forces known as drag and friction, which cause objects to come to a stop. Friction occurs between two surfaces touching each other. Drag is something similar but it occurs when a moving surface travels through a fluid.

Our atmosphere is made up by gas but for the physics calculations we need here there is no difference between this kind of gas and a fluid. So all non self-propelled objects moving around in our world are either slowed down by friction or by drag. The friction can be with the ground surface or any other surface they touch and the drag is normally due to surrounding fluid such as air or water. If you start looking up to the stars you will find that in space moving objects such as satellites seem to move continuously without slowing down. That is because of Newton's first law in combination with the absence of friction and drag.

Of course that is not totally true. Even in free space there are billions and billions of micro-small particles moving around which will collide with moving objects in space and influence their velocity by slowing them down.

#### ***Photonic Propulsion***

*Stars like the sun shoot billions of particles as well as photons into space every second. Such a particle stream is also called solar wind. Now rocket scientists would like to use the energy of those streams to power deep space probes. Therefore, those probes have a huge solar sail that is used as collision surface for the particle stream. The energy from the collisions will then push the probe forward.*

*But the main energy of this system is generated by the photons itself, not by the colliding ions. Light hitting a surface generates pressure. Only a pretty small pressure of course,*

*but in space with no friction and no drag this pressure is enough to propel a probe.*

*The idea of the solar sail is based on an experiment done by James C. Maxwell back in 1873. He found that light reflected by a mirror put a pressure on the mirror itself. This is backed up by the findings of Einstein that photons do have a mass.*

*In Practice this idea has some problems, though. First of all the effect would decrease with distance from the star. Second, the sail would need to be huge. One common example says that a sail with the area of 64.000 square meters would need a whole day to accelerate to a speed of 160 km per hour. But still, after three years it would sail with 160.000 km per hour making it three times faster than conventional NASA space probes propelled by conventional ion engines. At that speed it could reach Pluto within five years. Today, NASA needs twice the time to send a conventional probe to the outermost planet of our solar system.*

*The first solar sail orbit test failed in 2001 but still there are NASA programs that do research in this direction. And this little example shows that the so called "free space" is all but empty and free of particles causing collisions and thus friction and drag of some kind.*

But for the sake of simplicity in a video game you can assume that there is no drag and no friction in free space that has a noticeable influence on a space ship sized object – especially over short periods of time. But lets get back to earth's surface and discuss friction and drag, shall we?

### **2.3.3.1 Static Friction**

If you try to push an object that rests on the ground or any other surface you will find that the object won't move unless you push with some amount of force. Somehow, the object seems to resist against the push just as it does not want to be moved. And there is indeed a strange thing going on. As soon as you start pushing the object it will push back with a force of

equal magnitude but opposite direction. Hence it will remain at rest if you don't push harder. This force is due to static friction.

*The amount of friction does not depend on the size of the contact area between the two surfaces. If you have a box with different length for all three dimensions (width, height, and depth) it does not matter which one of the different sized sides of the box rests on the surface. The friction will always be the same.*

If an object rests on a surface its weight causes a pressure on the surface and there is friction between these surfaces. This frictional force attains its maximum value at the point just before the incoming force will make the object move. Figure 1 shows this situation. There is a total force  $\mathbf{F}$  exerted on an object. Due to the static friction the object experiences an opposite force  $\mathbf{F}_{SF}$  that will neutralize the incoming force  $\mathbf{F}$ . You can also see that the maximum magnitude of this static friction force is  $\mathbf{F}_{SF}^{\max}$ .

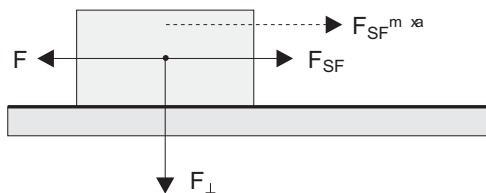


Figure 1: Force exerted on a body due to static friction.

As long as the total force  $\mathbf{F}$  exerted on the object is smaller or equal to  $\mathbf{F}_{SF}^{\max}$ , the object does not move. The force  $\mathbf{F}_{\perp}$  in Figure 1 is called the normal force, which is important to calculate the value of  $\mathbf{F}_{SF}^{\max}$  using the following equation:

$$\mathbf{F}_{SF}^{\max} = \mu_s \cdot \mathbf{F}_{\perp}$$

The normal force  $\mathbf{F}_{\perp}$  is the force that keeps the object in resting contact with the surface. This is normally the gravitational force.  $\mu_s$  is called the coefficient of static friction. Unfortunately, there's no way to obtain a correct value for a coefficient. The only way of getting an exact value for

such a coefficient is to measure it through experiment. In case of the coefficient of static friction you would put a test object onto an incline. Then you increase the incline and as soon as the object starts to slide you measure the angle of the incline. Figure 2 shows the setup for this experiment.

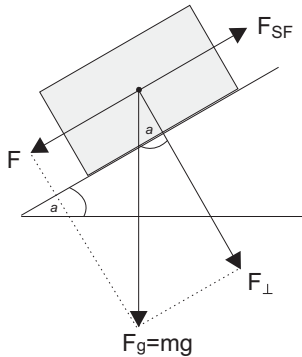


Figure 2: Measuring the coefficient of static friction.

From Figure 2 you can see that using the angle of the incline, you can calculate the normal force, which you can then use to calculate the frictional force:

$$F_{SF}^{\max} = F_g \cdot \sin \phi_{\max} \qquad F_{\perp} = F_g \cdot \cos \phi_{\max}$$

With this knowledge you can then rearrange the equation used to calculate the frictional force to be able to calculate the coefficient of static friction:

$$\begin{aligned} F_{SF}^{\max} &= \mu_s \cdot F_{\perp} \\ \Leftrightarrow \mu_s &= \frac{F_{SF}^{\max}}{F_{\perp}} = \frac{F_g \cdot \sin \phi_{\max}}{F_g \cdot \cos \phi_{\max}} \\ \Leftrightarrow \mu_s &= \tan \phi_{\max} \end{aligned}$$

Naturally, a single coefficient of static friction is only valid for the two specific materials involved. An object sitting on an incline made of a smooth material as opposed to a rough one will of course exert a different amount of friction. Finally, it all boils down to looking up coefficients of friction in tables once you know both materials involved. Then you need to

calculate the actual angle  $\phi$  of the incline, which lets you calculate the force due to static friction.

*Once there is a force that overcomes static friction, the object starts moving. As soon as the object is moving the static friction is no longer taken into consideration. That means not only is the sum of the total force and the frictional force used to calculate the acceleration of the object, but the whole total force without the static frictional force.*

### 2.3.3.2 Dynamic (kinetic) Friction

At the end of the preceding paragraph I told you that the force due to static friction vanishes as soon as an object has a velocity other than zero with respect to the surface it's resting. This does not mean that there is no longer any friction in play. Actually, the same equation will still be used. The only difference is that we are now talking about dynamic friction, which is also referred to as kinetic friction in some textbooks. Here is the equation:

$$F_{DF} = \mu_d \cdot F_{\perp}$$

The amount of dynamic friction depends on the coefficient of dynamic friction  $\mu_d$ . The meaning of this equation is that once an object is moving there needs to be a force  $F_x$  exerted on the object that is greater than the force  $F_{DF}$  due to dynamic friction to keep it going. Otherwise the dynamic friction would slow down the object to a stop where static friction takes over again. There is not much more to say about dynamic friction. You also need to look up in tables to find the coefficient  $\mu_D$  that is valid for the two involved materials. There is one interesting condition you should know if you want to guess your own coefficients. The coefficient of dynamic friction is always smaller than the coefficient of static friction.

$$\mu_d < \mu_s$$

### 2.3.3.3 Rolling Friction

The final type of friction I would like to discuss is called rolling friction. Just think of a driving car and its wheels. The wheels neither rest nor slide on the road under normal driving conditions. So there is neither static friction nor dynamic friction. Still you know that there is some kind of friction and this kind of friction is called rolling friction. I assume its equation will bore you to death, but anyway here it is:

$$F_{RF} = \mu_r \cdot F_{\perp}$$

Yet again you need to know another coefficient – the coefficient of rolling friction. But now there is an interesting fact. Most of the time rolling friction will occur on a sphere or a cylinder, naturally. In experiments physicians found that you can approximate the coefficient  $\mu_r$  by dividing the length of the contact area by the radius of the object:

$$F_{RF} = \frac{L}{r} \cdot F_{\perp}$$

With this equation you won't need look-up tables for the coefficient. Similarly to the types of friction mentioned above the rolling friction will prevent a sphere or cylinder from starting to roll or slow down, if the total force is below or above the force of rolling friction.

*The reason why car anti-lock breaks systems (ABS) work is because of the small coefficient of dynamic friction. If a car breaks and the wheels lock they start sliding. The resulting breaking force due to dynamic friction is less than the frictional force of turning wheels.*

Rolling friction is neither static nor dynamic friction. Once a wheel is turning, the contact point with the ground surface is not moving with respect to the ground surface as soon as the wheel reaches a stable rolling velocity. So the rolling friction is more of a static friction than a dynamic friction, but since the wheel is moving it can't be static friction. Physicians argue that the rolling friction is due to the deformation of the wheel that consumes energy and generates heat. This energy consumption is the reason why wheels will slow down and come to a stop.

### 2.3.3.4 Viscous Drag in Fluids

Now you have heard a lot about friction that exists between two solid surfaces that are either resting or moving with respect to each other. What happens if you have an object that is not resting, sliding or rolling across a solid surface? As long as this object is not located in free space it is located in a fluid. Even the air making up our atmosphere is such a fluid.

You all know from your own experience that a fluid such as air or water slow down the movement of an object similar to how friction does. In this case the force opposing the movement of the object is called viscous drag – or just drag which is, obviously, one kind of friction. Calculating the drag for an object is very simple even if the corresponding equation looks weird and quite complicated:

$$F_D = 0.5 \cdot C_w \cdot \rho \cdot A \cdot V^2 \quad \left[ \frac{\text{kg}}{\text{m}^3} \cdot \text{m}^2 \cdot \left[ \frac{\text{m}}{\text{s}} \right]^2 = \frac{\text{kg} \cdot \text{m}}{\text{s}^2} = \text{N} \right]$$

Hey wait. This looks strange at the first glance but it is by far not as complicated as it looks like. Lets start unrolling the equation from right to left.  $V^2$  is the squared velocity of the fluid hitting the object. That is nothing else than the squared velocity of the object with respect to the fluid. The variable  $A$  is the area of the surface that generates the drag, which is more or less the silhouette of the object which hits the fluid stream heads on. The Greek letter  $\rho$  (rho) is the density of the fluid. Finally, here you have  $C_w$ , which is the drag coefficient

*The drag force is always opposite to the velocity vector of the moving object. If the object does not move with respect to the surrounding fluid then there is no drag force at all.*

If you are interested in car or even in aircraft design you will know that the  $C_w$  value of an object depends on its shape. The more aerodynamically an object is, the smaller is the  $C_w$  value. Basically, the equation to calculate the drag says that in fluids there is a force exerted on moving objects. You can use this equation to calculate the drag exerted on objects moving

through air (air resistance), through water (water resistance), and any other fluid. So you can use the same equation for the drag in a racing car simulation as well as in a submarine video game.

To conclude this section about forces causing resistance to an object's movement take a look at Table 2. There you can find some typical drag coefficients for common shapes.

Object	Parachute	Circle	Human	Sphere	Typical Cars	F1 Car	Drop-shape
Cw	1.40	1.12	0.78	0.4	0,3-0.5	0.25	0.05

*Table 2: Typical drag coefficients.*

## 2.3.4 Buoyancy: Why Boats float and Elephants sink

Once upon a time in the third century before Christ there was a king called Hiero of Syracuse. Like all kings he used to wear a crown displaying his status and wealth. The royal goldsmith made this crown from the royal treasury. Hiero suspected that the goldsmith was trying to steal from the crown, literally speaking. The king was afraid that the goldsmith manufactured the crown using a cheaper alloy instead of the pure gold that he was given from the royal treasury.

Now the king needed to prove this speculation and find some evidence to hold against the goldsmith. It's quite obvious that you couldn't just put a sample from the crown into the gas chromatograph of your chemical laboratory back then. Furthermore, the crown must remain undamaged. So the king asked Archimedes, a noted Greek mathematician and natural philosopher, to solve this problem. Archimedes knew of course that alloy has less density than pure gold, but there was no way of determining the density of the irregular shaped crown. The density of an object is its mass per volume like this:

$$\rho = \frac{m}{V}$$

So the problem was to calculate the exact volume of the crown. When Archimedes tried to relax and to think about this problem while using the public bath he observed quite an interesting fact. If someone or something enters the tub then the level of the water raises. All of a sudden he knew how to solve the king's problem and ran naked through the town shouting "Eureka, Eureka". (I've found it, I've found it).

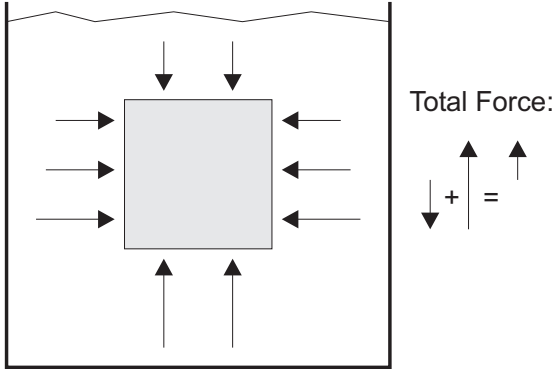
So, what did he find in the bath and how could this solve his problem? What Archimedes found is the principle of buoyancy. An object that is partially or wholly immersed in a fluid displaces a certain amount of water. Of course the amount of water displaced by this object equals the immersed volume of the object. Now, Archimedes could put the crown into a tub of water and measure the volume of the water being displaced by the crown. With the known mass of the crown he found that the density of the crown was less than the density of pure gold. Hence he could prove that the goldsmith was indeed trying to betray the king.

Well, this story is a good introduction to the topic of buoyancy but we are not yet finished. You know that objects immersed in a fluid displace the fluid by an equal amount of volume of the objects. As you will see later in this paragraph you can use Archimedes' trick to calculate whether or not a certain object floats, sinks, or even lifts up in the fluid. But first, we need to examine where the buoyancy force comes in play.

*Not only water and other liquids, like oil, are fluids. Strictly speaking, air is also a fluid that exerts a buoyant force on all objects. But since the density of air is comparably small you can treat it as non-existent.*

Due to the gravitational force, a fluid exerts pressure on objects inside the fluid. Roughly speaking you can say that an immersed object has to carry the weight of the fluid piled on top of it. But this is not only true for the fluid above the object. The pressure exerts a force from all sides onto immersed objects. That is yet again due to Newton's third law. Because the

immersed object displaces the water away from its surfaces the water in turn exerts an opposing force. But now the difference in density for different depths kicks in. Due to the fluid molecules piled on top of each other, density increases the deeper you go. Now take a look at Figure 3:



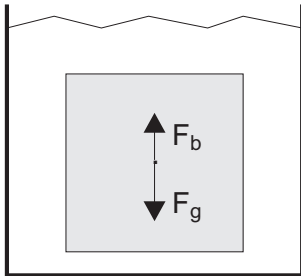
*Figure 3: Forces exerted on an immersed cube.*

The forces exerted on an immersed cubed increase with the augmentation in density. That means the downward forces on the upper side of the cube are smaller than the upward forces hitting the bottom of the cube. If you sum up all the forces, then the total force exerted on the cube– or any other immersed object for that matter – is always pushing the object upwards. Note that the forces hitting the sides of the cube have equal magnitude but opposing direction. Hence they cancel out each other.

*For a video game the change in density of fluids due to depth is not noticeable for small distances. Hence you can treat the density of your fluids as a constant value. In reality the density also depends on the temperature of the fluid.*

But hey, did I just say that buoyancy is always a positive force pushing all objects inside a fluid towards the surface? So why do objects like elephants sink in a fluid like water? Although the force due to buoyancy is an upward force, the net force exerted on an immersed object is of course not only buoyancy. You also need to take other forces into consideration – mainly the gravitational force. Figure 4 shows you the same cube with the

same buoyancy force, but now there is also the gravitational force indicated. As you can see the gravitational force is greater than the total buoyancy force. If you calculate the total or net force exerted on the cube by adding both force vectors you will see that the cube will sink in the fluid even if the buoyancy is an upward force.



*Figure 4: Net buoyancy force.*

With this knowledge at hand you can also find out when an object will float, sink or lift. What Archimedes found was that the volume of the water displaced by an object can be used to calculate the density of the object. The buoyancy force due to the displaced fluid is proportional to the weight of the displaced fluid. To conclude, that means if the immersed object weights more than the fluid weights it displaces then the object will sink in the fluid – that is because the gravitational force of the object is greater than the buoyancy force.

Now that you know the background of buoyancy you can start developing the equation to calculate the force due to buoyancy. All you need to know is yet again Newton's second law of motion:

$$F_b = m \cdot a$$

A force equals the object's mass times the acceleration. In this case the acceleration is the inverted gravitational acceleration  $g$  because you know that buoyancy is opposing the gravitational force pulling the object down.

$$F_b = m \cdot -g$$

Finally, you know that the mass of the displaced fluid equals the density of the fluid times the volume of fluid displaced by the object.

$$F_b = \rho \cdot V \cdot -g$$

And here you go. To calculate the force due to buoyancy acting on an object, you only need to know the density of the fluid, which you can look up in tables, as well as the gravitational acceleration and the volume of the object that is immersed. So the only thing you need to do at runtime of your program is to calculate the submerged volume of your object. With this volume at hand you can then calculate the buoyancy force pushing your object to the surface. This equation is all you need in the implementation of your physics engine but in the remainder of this section I will add some more background knowledge.

From this equation you can also see the rule when objects will float or sink. An object will sink if the following condition is met:

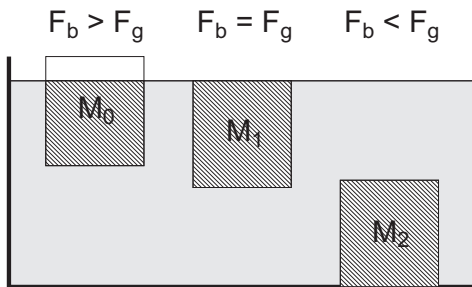
$$\begin{aligned} |F_g| &> |F_b| \\ \Leftrightarrow \rho_{\text{fluid}} \cdot V_{\text{fluid}} \cdot g &> \rho_{\text{obj}} \cdot V_{\text{obj}} \cdot g \\ \Leftrightarrow \rho_{\text{fluid}} \cdot V_{\text{fluid}} &> \rho_{\text{obj}} \cdot V_{\text{obj}} \\ \Leftrightarrow \rho_{\text{fluid}} &> \rho_{\text{obj}} \end{aligned}$$

If you look at the absolute values you can omit the sign of the acceleration term. Hence you can cancel out the acceleration term from the relationship. Next, the volume of the displaced fluid equals the volume of the completely immersed object so you can cancel that out as well. As result you can see what most of you will already know. An object sinks in a fluid if its density  $\rho$  is greater than that of the fluid. The good news is that you don't need to take care of that. We are only looking for the buoyancy force of the object. It will be combined with the gravitational force automatically when updating the object in our program.

*Submarines work by simply changing their density. In general, a submarine is balanced to swim partially immersed like a ship. If it wants to dive it fills ballast tanks with water venting out the air of the tanks. Now the overall density of the*

*submarine is greater than that of the water and it starts to sink. If the submarine wants to surface it pumps compressed air into the ballast tanks venting out the water to decrease the overall density of the submarine.*

Still this relationship indicates a cheap way out for you. You could simplify the buoyancy calculation by only comparing the density of your object with the density of the fluid. Then just let it sink with a certain speed if it's denser than the fluid. There are two reasons why this might not always be the best approach. First of all, how would you calculate the speed for the object to sink or to rise up inside the fluid? The denser the object the faster it would sink, but you could only approximate the speed. The second and more interesting question is how do you want to deal with partially immersed objects?



*Figure 5: Three solid cubes with different density.*

Figure 5 shows different objects that are partially and fully immersed. Think of an object that will float in water because it is as dense as water. Now put the object onto the water surface and you will see that it will start to sink. Why is that? Because if the object is put right on top of the water surface only a small part of its overall volume is immersed. Hence the buoyancy force is pretty small while the gravitational force will already exert its full power. While the object sinks deeper, the overall immersed volume increases and the buoyancy force increases accordingly. Finally, the object is fully immersed and only now the buoyancy force has the same magnitude as the gravitational force. Hence the object starts to float in this position. Now think of an object that has a slightly smaller density than water. Similarly it will start to sink when put onto the water surface, but the

buoyancy force will already equal the gravitational force before the object is fully immersed. That means some of the object's volume will stay immersed even if density says that the object does neither sink nor float but lift up instead. In Figure 5 the left-most cube has a smaller density than the fluid. But if the object is only partially immersed it does not displace enough water to let it go up any longer. Hence it stays in equilibrium as soon as the amount of displaced water weights as much as the whole object.

*In the absence of a gravitational force there is no change in density throughout the fluid. Hence the pressure on an immersed object is constant all over its surface and there is no buoyant force at all.*

Okay, one last thing before we move on to the next force that can influence your point mass objects. Buoyancy will not always push things upwards. Remember that your virtual world could be weird and not obey the normal law of physics. Just like drag is always opposing velocity the buoyancy fore is always opposing the gravitational force. If your gravity works upwards instead of downwards then buoyancy will push objects in fluids downwards. :-)

## 2.3.5 Lift: How do planes fly

In the last section, we discussed buoyancy in detail. We could do the same for the force that creates a lifting effect but even though this is a favorite topic of mine I will not do that. Here's why: A force that creates a noticeable lift for your physics objects can only exist for surfaces that are constructed and shaped in a special way. Yes, I'm talking about airfoils. Furthermore, lift depends totally on this object to move with a relatively high speed with respect to the stream hitting the object in a special angle. What I'm saying is that if your object does not have this special shape, or if it does not move with at least 150 mph for a small plane, or if it does not hit the incoming stream within a rather small angle with its leading edge, there would be no lift at all.

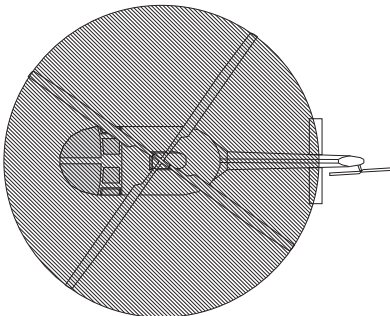
That is why you can assume that only objects like airfoils generate a noticeable lift force. This means that you should only examine this topic in details if you are going to implement a flight simulator. Nonetheless I want to give you the equation that lets you calculate the lift so we can integrate it into our implementation. You never know when you're going to program you next flight simulator, right?

*From the preceding section you know that air is just a fluid like any other. So if there is buoyancy in air then there is of course also lift in fluids like water. If you have something like a submarine featuring airfoil-like stabilizers, they will also generate lift.*

To make it quick I will just show you the equation that is used to calculate the lifting force for a fast moving object:

$$F_L = 0.5 \cdot C_L \cdot \rho \cdot A \cdot V^2 \left[ \frac{\text{kg}}{\text{m}^3} \cdot \text{m}^2 \cdot \left[ \frac{\text{m}}{\text{s}} \right]^2 = \frac{\text{kg} \cdot \text{m}}{\text{s}^2} = \text{N} \right]$$

Does that ring some bells? This equation is the same as the one for viscous drag except for the different coefficient.  $V^2$  is the squared velocity of the free stream hitting the object. That is nothing else than the squared velocity of the wind with respect to the object. The variable  $A$  is the area of the surface that generates lift. In case of an aircraft that is roughly its wing surface, but other parts of the aircraft such as the fuselage can contribute to the lift as well.



*Figure 6: For a helicopter the lift area is the rotor disk area.*

$\rho$  is the density of the fluid, namely the density of the air. Finally,  $C_L$  is what we aircraft engineers call the lift coefficient. The lift coefficient is a dimensionless value that allows the comparison of lift incurred by different sized and different shaped bodies. This lift coefficient can have positive values indicating lift as well as negative values representing down force. The lift coefficient is a unique characteristic of an airfoil or a lifting surface in general and depends on the shape of the surface. As you can see in the equation a higher lift coefficient causes a greater lift. So the next logical question is how do you calculate the lift coefficient for a lifting surface? The answer is you can't.

Experienced aircraft engineers can guess lift coefficients for an airfoil by comparing it to similar shapes with known lift coefficients. But even today the lift coefficients for new airfoils are retrieved by real wind tunnel experiments with small-scale high-precision models. Additionally, the final lift coefficients for an airfoil will be collected by test flights with the first real prototypes. And yes, there are multiple lift coefficients for a single airfoil because the lift coefficient depends on the angle with which the free-stream velocity hits the lifting surface's leading edge. The hobby pilots among you know this angle as the angle of attack or AOA.

*If the AOA gets too big an aircraft suffers a stall. The lift coefficient drops to 0 or even negative values. Then the aircraft starts falling like a stone. The same happens if the AOA gets negative due to the free stream hitting the surface from below. This is why airfoils mounted on aircrafts are not aligned with the aircraft's longitudinal axis but are inclined a bit so that the AOA is a small positive one in steady flight.*

Another thing you can see from the equation for the lift force is that the velocity of the aircraft has a big influence on the lift. That means even an airfoil with rather bad lift coefficients can generate a decent lift if the aircraft goes fast enough. This is interesting for supersonic flights, for example, where the airfoil must meet other conditions as well to enable a steady flight.

### **Simulating Helicopters**

*The lift force equation presented in this section can also be used to calculate the lift generated by a helicopter's main rotor. The lifting surface is the rotor disk area, or the area of the circle through which the rotor blades turn.*

*However, this is only true if the helicopter is moving forward with at least some speed. For a slow flight or even a hovering helicopter this equation does not hold true any longer since the lift force would then be 0 according to this equation.*

*Unfortunately there is no single, simple equation that can be used to calculate the exact lift for a hovering helicopter. But you can think of the rotor to be an airplane propeller that has been turned 90 degrees. The rotor sucks in air from above and accelerates it downwards while the air goes through the rotor disk creating. So there is a force acting downwards on the air stream. Hence, due to Newton's third law, there is a force that acts in the opposite direction. And that force is lift.*

*If you want to program a helicopter simulator I would suggest faking the lift force for hovering situations. To keep the helicopter steady you need to have a lift force that is equal to the gravitational force. Then the pilot should be able to increase and decrease this lift force a bit to be able to go up and down, respectively.*

Now you know enough to calculate the lift for your objects. You only need to know the density of the fluid (air), the area causing the lift on your object, the velocity of the object, as well as its current lift coefficient.

*Try googling for lift coefficients. In the internet you can find a lot of sources with lift coordinate tables for all kinds of aircrafts – e.g. NASA has numerous papers documenting results from flight experiments.*

## 2.4 Implementation

If you are still with me at this point you can give yourself a pat on the back. You have just mastered a hell of a lot of basic mechanics and this physics foundation is enough to start implementing our first game dynamics simulation. In the following chapters we will add more and more functionality to our source code repository and by the end of this book you will have a small but complete and combat ready game dynamics engine at your disposal.

You now know how to calculate the acceleration, velocity, and change of position of a physics object. This functionality will now move into a class called `PointMass`. When we start dealing with rigid bodies you will find that a lot of functionality used by the point mass can be reused for the rigid body. Being a bit lousy in definitions, one can say that a rigid body is a point mass that has a volume other than zero. So it is tempting to say you can derive a class for rigid bodies from the `PointMass` class later on, but of course this wouldn't be good object oriented design since you can't really say that a rigid body is some kind of point mass. At least not if a physician is present. That is why I decided to introduce another basic class called `DynamicsObject`. This class contains all basic functionality such as setting or accessing forces exerted on an object. Then we can derive the `PointMass` class from the `DynamicsObject` class and add the few functions that are rather unique to a point mass. Later in this book we will derive a class `RigidBody` from the `DynamicsObject` class as well.

Figure 7 shows a UML diagram with the two classes we are going to implement now. Note that I have omitted some attributes and member function of the classes for the sake of clarity. If you are not familiar with UML diagrams then please refer to the excursion box at the end of this section.

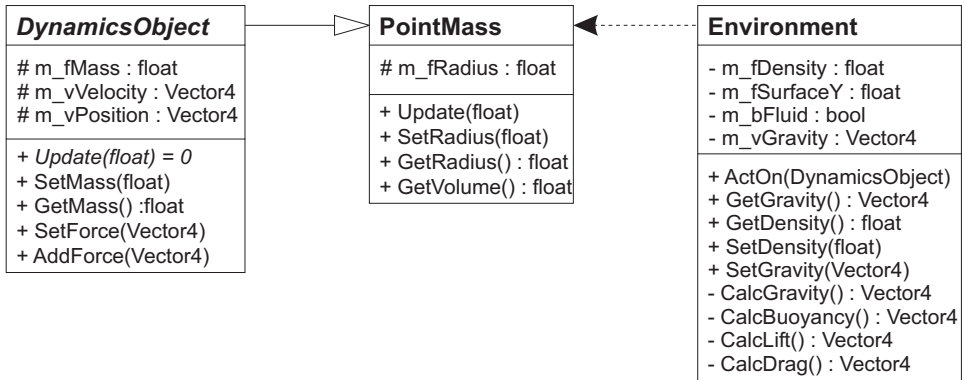


Figure 7: UML diagram of our current physics system.

Basically the DynamicsObject class handles adding forces you want to exert on the object. It also takes care of keeping the units of the values to a common system to avoid the apples and oranges issue. Then, the PointMass class is rather simple and short, as you will see in a minute. If you're asking yourself why the PointMass class has a radius and a volume you can find the answer in the next paragraph. Figure 7 also shows a class called Environment. Because a lot of forces exerted on physical objects are more or less due to the natural environment, I decided to implement those forces in such a class.

There are four forces which the environment can exert on an object: Drag, Gravity, Lift, and Buoyancy. For now you don't need to care about the environment class. First, we will discuss the DynamicsObject and the PointMass class.

### UML Diagrams

The acronym UML stands for Unified Modeling Language which in turn is a standard of data representation for object oriented computer software. One major part of UML is the class diagram notation that is used to represent static elements of a design such as classes and relationships. The other part of UML deals with dynamic elements such as state machines and messages. There are even software tools that let you specify your design using UML. Once you are done

with your design, such a tool can create a complete source code project using your language of choice and generates all the files, definitions, prototypes, and functions. The only thing you are then required to do is to write the function bodies between the empty brackets.

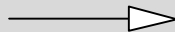
Because UML diagrams are quite handy I want to introduce you to the major notations of UML class diagrams. The fundamental base of a class diagram is of course an icon to represent a class:



This icon has three different parts. The upper part contains the class name, the middle part contains a list of the class' attributes, and the lower part contains the list of the class' functions. On the right side you can see the modifiers that show the visibility (public, protected, and private) of attributes and functions. This modifier should precede the name of the corresponding class member.

If the class name is written in italic it means that the class is abstract. If the name of an operation is written italic it means that the corresponding function is pure virtual.

To show the inheritance hierarchy UML uses lines connecting two classes where the end of line that connects to the parent class show a hollow arrow like this one:



A weak dependency between two classes means that one class knows about the other class because e.g. it is using this class as a parameter for one or more of its operations. UML shows such a weak dependency by connecting both classes with a dotted line. The line starts from the class that knows the other one and ends with a solid arrow like this one:



*And for now you should all you need to know at the moment to understand the UML diagrams I'm presenting throughout this book.*

## 2.4.1 Simulating the Point Mass

Actually, a point mass does not have a radius and hence it does not have a volume. It's really just an infinite small point that has a mass. This would be enough so simulate simple dynamics such as exerting forces on the point mass. But I would like to fake the visual impression here by using a radius for the point mass so that our implementation has a real volume associated with it. The main reason for this illogical design decision was the buoyancy.

The buoyancy force can only influence objects that displace fluid, but to be able to displace fluid an object needs to have a volume. That is why I decided to let our `PointMass` class have a volume. It just makes the demo application in this chapter more interesting. Otherwise, the only interesting thing you can do with point masses is simulating particles and particle systems. So keep in mind that point masses should not have a volume associated with them if you start designing your own dynamics engine.

*The `PointMass` class presented in this section uses a radius and hence a volume associated with the object. So it is rather a spherical rigid body instead of a point mass. We currently lack the ability to have angular velocity to change the orientation of the object, but for a complete rigid body we would need that. Just keep in mind that point masses usually don't have a volume at all.*

And finally here it is – this chapter's first source code. Let me introduce you to the `DynamicsObject` class, which is the base class for all of our dynamics objects. Note that this class is an abstract class because it has a number of pure virtual member functions. Take a look at the class declaration first, and then we can discuss the implementation.

```
class DynamicsObject
```

```

{
public:
    DynamicsObject();
    virtual ~DynamicsObject() { ; }

    virtual void          SetPosition( const spg::Vector4 &v, Unit u );
    virtual void          SetLinearVelocity(const spg::Vector4 &v, Unit u );
    virtual void          SetMass( float f, Unit unit );
    virtual void          SetForce( const spg::Vector4 &fforce, Unit u );
    virtual void          AddForce( const spg::Vector4 &fforce, Unit u );
    virtual void          AddLinearVelocity(const spg::Vector4 &v, Unit u );
    virtual void          SetLiftingSurface(float fArea, const
                                                spg::Vector4 &vNormal, Unit u);

    virtual float         GetMass( Unit unit )const;
    virtual float         GetSpeed( Unit unit )const;
    virtual spg::Vector4  GetForce( Unit unit )const;
    virtual spg::Vector4  GetLinearAcceleration( Unit unit )const;
    virtual spg::Vector4  GetLinearVelocity( Unit unit ) const;
    virtual spg::Vector4  GetPosition( Unit unit ) const;
    virtual spg::Vector4  GetPositionLast( Unit unit ) const;
    virtual float         GetVolume( Unit unit )const;
    virtual float         GetLiftArea( Unit unit ) const;
    virtual spg::Vector4  GetLiftingSurfaceNormal() const
        { return m_vLiftNormal; }

    // pure virtual functions
    virtual void          Update( float fElapsedSec )=0;
    virtual float         GetVolumePercentBelowY( float fY )const=0;
    virtual float         GetDragArea()const=0;
    virtual float         GetDragCw()const=0;
    virtual float         GetLiftCl()const=0;

protected:
    float                m_fMass;           // kg
    float                m_fVolume;        // meter^3
    float                m_fSpeed;         // m/s
    float                m_fElapsed;       // from last frame
    float                m_fLiftArea;      // m^2
    spg::Vector4         m_vPosition;      // meter
    spg::Vector4         m_vPositionOld;   // meter
    spg::Vector4         m_vLiftNormal;    // lifting surface normal
    spg::Vector4         m_vF;            // forces acting N
    spg::Vector4         m_vV_lin;        // linear velocity m/s
    spg::Vector4         m_vA_lin;        // linear acceleration m/s^2
};
/*-----*/

```

Actually, there is no need to discuss the implementation of the non-pure virtual functions. All of the functions are just accessor methods to the

corresponding member variables. To give you an example I will show you one implementation and then you will know them all:

```
void DynamicsObject::SetPosition( const spg::Vector4 &v, Unit unit )
{
    m_vPosition = Convert::Length_From_To( v, unit, UNIT_LENGTH_M );
}

spg::Vector4 DynamicsObject::GetPosition( Unit unit ) const
{
    return Convert::Length_From_To( m_vPosition, UNIT_LENGTH_M, unit );
}
```

Here you can see that the class uses fixed units for all of its members to make sure that all forces you add for one object use the same unit. The user is free to hand in or request all member variables in all available units. The accessor member functions will automatically convert the units accordingly.

*In the real source code implementation of this chapter I have used default parameters for all of the `Unit` parameters. The default values correspond to the unit a particular member variable uses internally. You can see those units from the comments in the class declaration.*

With the base class out of the way we can now take a look at the `PointMass` class. Because most of the accessor functions that set or retrieve the member variables are implemented in the base class there is not much left to do in the derived class.

```
class PointMass : public DynamicsObject
{
public:
    PointMass( float fRadius, Unit unit = UNIT_LENGTH_M );
    virtual ~PointMass() { ; }

    virtual void Update( float fElapsedSec );
    virtual void SetRadius( float fRadius, Unit unit = UNIT_LENGTH_M );

    virtual float GetRadius( Unit unit = UNIT_LENGTH_M ) const;
    virtual float GetVolumePercentBelowY( float fY ) const;
    virtual float GetDragArea() const;
    virtual float GetDragCw() const;
```

```

        virtual float    GetLiftCl() const;

protected:
        float           m_fRadius;           // meter
};
/*-----*/

```

As you can see the only additional member variable used in this derived class is the radius of the point mass. You should consider removing this radius from the point mass class later and come up with a rigid body sphere class instead. The next paragraph will introduce you to the implementations of the member function of the PointMass class starting with the accessor functions.

### 2.4.1.1 Accessor Functions

Well, there is not much to tell about most of the accessor functions. The volume of a sphere can be calculated with the following equation, which needs to be done each time you change the object's radius:

$$V_{\text{sphere}} = \frac{4}{3} \cdot \pi \cdot \text{radius}^3$$

So here is the corresponding function body:

```

// set radius of point mass to change initial constructor settings
void PointMass::SetRadius( float fRadius, Unit unit )
{
    m_fRadius = Convert::Length_From_To( fRadius, unit, UNIT_LENGTH_M );

    if ( m_fRadius < 0.0f ) m_fRadius = 0.0f;

    m_fVolume = 1.33f * Const::PI * (m_fRadius*m_fRadius*m_fRadius);
} // SetRadius
/*-----*/

```

Next, I want to show you how to get the drag area for the point mass. As you know drag acts opposite to the velocity vector of an object. Then the area causing the drag you need to use in the drag equation is the silhouette area as seen from an observer located ahead of the object with respect to the velocity vector. Since our point masses are spheres we can

assume this area to be a circle with the sphere's radius. Of course this is not totally true but it is good enough for a video game. Here's how you calculate the area of a circle:

$$A = \pi \cdot \text{radius}^2$$

Here is the corresponding function body:

```
float PointMass::GetDragArea() const
{
    return ( Const::PI * (m_fRadius*m_fRadius) );
} // GetDragArea
/*-----*/
```

Out of the remaining functions there is another one of interest remaining. You will already have noticed the base class' DynamicsObject::GetVolumePercentBelowY member function. The purpose of this function is to calculate the volume of the object that is below a horizontal plane at a certain height y-value. Figure 8 shows a situation where a sphere intersects a horizontal plane and how the plane separates the volume of the sphere.

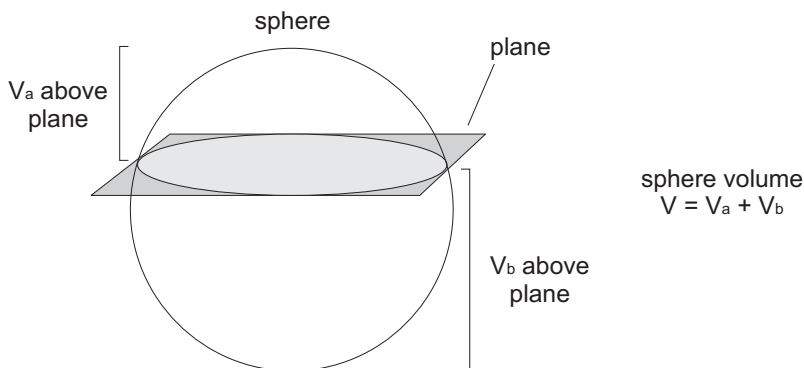


Figure 8: Volume separation of a sphere with respect to a plane.

The purpose of the GetVolumePercentBelowY() function is to calculate the volume of the object that is below the horizontal surface. This task is not a trivial one unless you want to invest a lot of calculation power. So lets discuss the reason why you need this function first. If you want to calculate the buoyancy of your DynamicsObject instance you need to know the

immersed volume of the object. Unless the object is neither wholly immersed nor wholly outside of the fluid you need to calculate the exact immersed volume. The horizontal plane is the fluid's surface and the volume below the plane's y-value (height) is the immersed volume of the object.

With that made clear we still have the problem of calculating the immersed volume. Instead of using time consuming exact calculations we can live with a fast approximation here. My implementation suggests that the volume of the sphere is equally distributed over its height. That is of course not true, as the volume gets exponentially smaller if you get closer to the poles of the sphere. As you can see in the demo of this chapter the approximation looks good which let us stand to our maxim "*If it looks good, it is good*".

Here are the missing function bodies:

```
float PointMass::GetVolumePercentBelowY( float fY ) const
{
    // for sake of simplicity assume an equal distribution of the
    // volume over the sphere - which is of course not true.
    float fMinY = m_vPosition.GetY() - m_fRadius;
    float fD = 2.0f * m_fRadius;

    if ( fMinY >= fY )
    {
        // totally above Y
        return 0.0f;
    }
    else if ( (fMinY + fD) <= fY )
    {
        // totally below Y
        return 1.0f;
    }
    else
    {
        // partially above and below Y
        return ( (fY-fMinY) / fD);
    }
} // GetVolumePercentBelowY
/*-----*/

float PointMass::GetRadius( Unit unit ) const
{
    return Convert::Length_From_To( m_fRadius, UNIT_LENGTH_M, unit );
} // GetRadius
/*-----*/
```

```

float PointMass::GetDragCw() const
{
    return 0.45f;
} // GetDragCw
/*-----*/

float PointMass::GetLiftCl() const
{
    return 0.0f;
} // GetLiftCl
/*-----*/

```

The drag coefficient is 0.45, which corresponds to a sphere. The lift coefficient is set to 0, which is true because a sphere has no real lifting surface. You should return 0 for the lift coefficient for all of your from DynamicsObject derived classes unless the object is a wing surface or has a wing surface. We should ignore the lifting issue for the next chapters because we don't want to deal with aircrafts yet.

### 2.4.1.2 Updating a Point Mass

As we approach the end of this chapter we are about to bring the theory we learned to conclusion. The updating function of the PointMass class will calculate the acceleration of the object based on the total force exerted on it. The linear velocity is updated with the acceleration times the elapsed seconds which effectively integrates the acceleration to get the change in velocity. Finally, you can use the new linear velocity vector to update the position of the object by integrating the time step with the linear velocity to get the change in position.

```

// update all forces and accelerations acting on the point mass
void PointMass::Update( float fElapsedSec )
{
    // store last known position
    m_vPositionOld = m_vPosition;

    m_fElapsed = fElapsedSec;

    // update linear acceleration:
    // F = m*a <=> a= F / m ( lbs / kg )
    m_vA_lin = m_vF / m_fMass;

    // update linear velocity:

```

```

// V = V + A * TimeStep ( m/s^2 * s = m/s )
m_vV_lin += m_vA_lin * fElapsedSec;

// update position of point mass:
m_vPosition += m_vV_lin * fElapsedSec;

// calculate speed
m_fSpeed = m_vV_lin.GetLength();

// reset counters
m_vF.Set( 0.0f, 0.0f, 0.0f );
} // Update
/*-----*/

```

Note that the function will also store the current speed of the object, which is the magnitude of the velocity vector. It will also set the force vector to zero to indicate that all forces exerted on the object during the last frame are now taken care of and won't influence the object any longer.

If you have a constant force exerted on the object such as an engine propelling it then you have to reapply it to the object each frame.

*For a more comprehensive implementation of a game dynamics engine you should consider writing a class that represents a force. Then your dynamics objects should be able to store constant forces to avoid the need to reapply them each frame. When we start dealing with rigid bodies you will see that you will also need a point of application for each force. This could also be part of a class *Force*.*

## 2.4.2 Simulating the Environment

Next in line is the Environment class. Up to now we have implemented the PointMass class, which is basically nothing more than a class which can accept forces exerted onto the corresponding physics object. We are still lacking the ability to calculate a force that should act on such an object.

Well, that is the purpose of the Environment class. It can calculate the four main forces that exist in nature and that will influence all objects moving

inside a certain environment. You already know the fundamentals of this class from the UML diagram in Figure 7.

```

class Environment
{
public:
    Environment();
    virtual ~Environment();

    void          ActOn( DynamicsObject &MyObject, float fInfluence );
    void          ActOn( DynamicsObject *pMyObject, float fInfluence );

    // accessor functions
    bool          IsFluid() const { return m_bFluid; }
    void          SetGravity( const spg::Vector4 &gF, Unit unit );
    void          SetDensity( float f, Unit unit );
    spg::Vector4 GetGravity( Unit unit )const;
    float        GetDesity( Unit unit )const;
    float        GetSurfaceY() const { return m_fSurfaceY; }
    void          SetFluid( bool b, float fSurfaceY=0.0f )
        {
            m_bFluid = b;
            m_fSurfaceY = fSurfaceY;
        }
private:
    bool          m_bFluid;           // fluid (else atmosphere)
    float         m_fDensity;         // kg/m^3
    float         m_fSurfaceY;       // units
    spg::Vector4 m_vGravity;         // m/s^2

    spg::Vector4 CalcBuoyancy_N( float fVolumeM3 );
    spg::Vector4 CalcGravity_N( float fMassKg );
    spg::Vector4 CalcDrag_N( const spg::Vector4 &vV_Dir, float fV,
        float fA, float fCw );
    spg::Vector4 CalcLift_N( const spg::Vector4 &vV_Dir,
        const spg::Vector4 &vLiftSurfaceNormal,
        float fV, float fA, float fCl );
};
/*-----*/

```

I need to add a few comments about the environment being a fluid or not. Strictly speaking even the air is a fluid, but for the purpose of this implementation an environment that is a fluid will automatically have a buoyancy force exerted on the objects it affects. Since the buoyancy in air is not noticeable due to its comparably small density, you should not mark your environment instance that is used for the normal air as fluid.

If you have a fluid, you need to set the y-value, which is the height of the fluid's surface in world coordinates.

*For the sake of simplicity I'm using a single float value to represent the volume of a fluid. That is enough for this chapter, but this starts to fall apart as soon as the gravity does not act downwards or as soon as you want to limit the volume of the fluid to something other than just occupying the whole half space of the complete virtual world below the given y-value.*

*A better design would include a real plane that represents the fluid's surface and an even better design would let you specify an arbitrary number of planes enclosing the volume that is occupied by the fluid. Then there is no such thing as a single surface and can simply test your object for intersection with the n-sided frustum of planes.*

### **2.4.2.1 Acting on an Object**

The only reason why we need to take an environment into consideration is that the environment exerts forces on objects that are at least partially located inside the environment. From the prototype of the `Environment::ActOn` function you can see that the parameter is the object onto which the environmental forces should be exerted. There is also a second parameter called `fPercentage`; this parameter specifies a value between 0.0 and 1.0 that says how much impact the environment has on the object.

This can be used for when you have an object that is partially immersed in a water environment while its remaining volume is located inside an air environment. If the object is immersed with 20 % of its volume, you would use a value of 0.2 for the function call of the water environment acting on the object. Then you would send the object into the air environment, which will act on it with a value of 0.8.

The rest of the function is quite simple because it will only call four sub functions that calculate the actual forces. Here is the function:

```

void Environment::ActOn( DynamicsObject &MyObject, float fPercentage )
{
    ActOn( &MyObject, fPercentage );
}
void Environment::ActOn( DynamicsObject *pMyObject, float fPercentage )
{
    if ( pMyObject )
    {
        // get speed and normalized velocity direction
        spg::Vector4 vV_Dir;

        if ( pMyObject->GetSpeed() != 0.0f )
        {
            vV_Dir = pMyObject->GetLinearVelocity()
                / pMyObject->GetSpeed();
        }

        spg::Vector4 vForces = CalcGravity_N( pMyObject->GetMass() );

        // in fluids add buoyancy
        if ( m_bFluid )
        {
            vForces += CalcBuoyancy_N( pMyObject->GetVolume() );
        }

        // drag force if object is moving
        if ( pMyObject->GetSpeed() != 0.0f )
        {
            vForces += CalcDrag_N( vV_Dir, pMyObject->GetSpeed(),
                pMyObject->GetDragArea(),
                pMyObject->GetDragCw() );
        }

        // lift force if lifting area
        if ( pMyObject->GetLiftArea() > 0.0f )
        {
            vForces += CalcLift_N( vV_Dir,
                pMyObject->GetLiftingSurfaceNormal(),
                pMyObject->GetSpeed(),
                pMyObject->GetLiftArea(),
                pMyObject->GetLiftCl() );
        }

        pMyObject->AddForce( vForces * fPercentage );
    }
} // ActOn

```

```
/*-----*/
```

As you can see certain forces depend on certain conditions. If the environment is not marked as being a fluid then the buoyancy force is not applied. Lift only occurs if there is a lifting surface area greater than 0 and drag can only exist if the object has an absolute speed greater than 0. The last line of the function instructions then adds the total force of the environment to the object. Note how the total force is scaled with the requested percentage value.

## 2.4.2.2 Calculating the Forces

With the knowledge about basic physics and mechanics you gained in the first section of this chapter, implementing the forces an environment exerts on an object is as easy as it gets. So I don't think there is any need to comment the following implementations.

```
spg::Vector4 Environment::CalcBuoyancy_N( float fVolumeM3 )
{
    if ( m_bFluid )
    {
        // F = rho * V * g
        return ( m_vGravity * (m_fDensity * fVolumeM3) * -1.0f );
    }
    else return spg::Vector4( 0.0f, 0.0f, 0.0f );
} // CalcBuoyancy_N
/*-----*/
```

```
spg::Vector4 Environment::CalcGravity_N( float fMassKg )
{
    // F = m * a = m * g ( kg*m/s^2 = N )
    return ( m_vGravity * fMassKg );
} // CalcGravity_N
/*-----*/
```

```
spg::Vector4 Environment::CalcDrag_N( const spg::Vector4 &vV_Dir, float fV,
                                     float fA, float fCw )
{
    spg::Vector4 vcDrag( 0.0f, 0.0f, 0.0f );

    if ( fV == 0.0f ) // no speed no drag
    {
        return vcDrag;
    }
    else
```

```

    {      // drag is opposing velocity
          vcDrag = vV_Dir * -1.0;

          // Fd = 0.5 * Cw * q * A * v^2
          float Fd = 0.5f * m_fDensity * (fV*fV) * fA * fCw;
          return (vcDrag * Fd);
        }
} // CalcDrag_N
/*-----*/

spg::Vector4 Environment::CalcLift_N( const spg::Vector4 &vV_Dir,
                                     const spg::Vector4 &vLiftSurfaceNormal,
                                     float fV, float fA, float fCl )
{
    spg::Vector4 vLift( 0.0f, 0.0f, 0.0f ), vTemp;
    spg::Vector4 vDrag = vV_Dir * -1.0;

    if ( fV == 0.0f )      // no speed no lift
    {
        return vLift;
    }
    else
    {      // lift is perpendicular to drag
          vTemp.Cross( vDrag, vLiftSurfaceNormal );
          vLift.Cross( vTemp, vDrag );
          vLift.Normalize();

          // Fl = 0.5 * Cl * q * A * v^2
          float Fl = 0.5f * m_fDensity * (fV*fV) * fA * fCl;
          return (vLift * Fl);
        }
} // CalcLift_N
/*-----*/

```

When you think about it then it is actually ridiculous: To implement your first cool game dynamics demo, you basically don't need more than the four functions shown above as well as the `PointMass::Update` function. All in all that's nothing more than just two dozens lines of source code. Now try to tell me that programming game dynamics is not as easy as eating pancakes. :-)

## 2.4.3 What about Friction?

Now take a deep breath. We are almost done for this chapter and you are really close to implement your first game dynamics simulation. With the

Environment class and the PointMass class we have everything we need to have a nice demo of balls thrown into a pool of water.

You might have noticed that we did not implement friction. For now I decided to ignore friction. Normally, it should be part of the DynamicsObject class; at least as a pure virtual member function enforcing all derived classes to implement friction. Try to remember the situations in which friction occurs. You need to have two objects touching each other or even colliding with each other. At the moment we have no means of calculating a colliding contact between our dynamics objects and we would only need friction in such a case. That is why I did not include its implementation in this chapter. Since we discussed the basic forces in this chapter I thought it would be good to let you know about the theory of friction. We will later revisit and implement friction.

## 2.5 Demo Application: Swimming Pool

After mastering all this physics stuff, we want to satisfy ourselves by seeing the physics calculations in action. Hence I decided to implement a small demo that will show off everything we have developed so far. The demo features a small room where the lower half of the room is filled with water. Then there are three spheres (read point masses) that drop into the water from the ceiling. The demo application will do nothing else than just put the three spheres below the ceiling of the room. Then the three spheres are just updated with the two environments existing inside the room. The first environment is the air in the upper half and the second environment is the water in the lower half.

When the demo application runs the environments will exert gravity and drag in the air and in the water as well as buoyancy in the water. You can use the m key to toggle through the three spheres and then use the 1 and 2 and the 3 and 4 keys to decrease and increase the mass and the radius of the current sphere, respectively.

For rendering, input, and audio (later) I use the **Sipogen** engine components **spg**, **spgaurd**, and **spgi**.

Basically, the most comfortable way to render something with this engine for prototyping purpose is to derive your own class `MySipogenApp` from the **Sipogen** base class `CEngineApplication`. By looking at the source code of the demo application you should have no problems understanding what's going on during the initialization and rendering phases.

The first one of the two most interesting functions from the demo application is the `MySipogenApp::MyInit` function. This is called after initializing the engine and it contains the setup of our physics objects as you can see:

```
bool MySipogenApp::MyInit()
{
    [...]
    // air environment
    m_EnvAir.SetDensity( spgdyn::Const::DENSITY_AIR_KGM3 );
    m_EnvAir.SetGravity( spg::Vector4(0,-spgdyn::Const::GRAVITY_MS2,0));
    m_EnvAir.SetFluid( false );

    // water environment
    m_EnvWater.SetDensity( spgdyn::Const::DENSITY_WATER_KGM3 );
    m_EnvWater.SetGravity( spg::Vector4(0,-spgdyn::Const::GRAVITY_MS2,0));
    m_EnvWater.SetFluid( true, 0.0f);

    InitPointMasses();

    GenerateUnitCube(m_Pool, m_PoolInd, true, spg::Vector4(0.0f,1.0f,0.0f),
                    8.0f, 8.0f, 8.0f, 0xffffffff, 0xffdddd, 0xffaaaa);

    GenerateUnitCube( m_Water, m_WaterInd, false, spg::Vector4(0.0f,
                      SEA_FLOOR/2.0f,0.0f), 8.0f, -SEA_FLOOR, 8.0f,
                      0x775555ff, 0x770000ff, 0x77aaaaff );
    [...]
}
/*-----*/
```

The `MySipogenApp::InitPointMasses` function will create a STL vector of three `PointMass` instances and set their starting position, mass, and radius. The `MySipogenApp::CreateUnitCube` function does exactly what it name suggests. It creates a cube with width, height, and depth of 1.0. This is a handy helper function that delivers the vertex data and the index data in

the first two parameters. The second parameter says if the cube should have inward looking faces. The fourth parameter defines the center point of the cube. Next, there are three optional parameters to change the width, height, and depth of the cube, followed by three colors that should be used for the three pairs of opposing faces.

With this nice function you can easily create a cube that can be rendered with the **Sipogen** render device. In this case one cube with inwards looking faces is used for the room and another cube with outwards looking faces is used for the water inside the room. Note the hexadecimal alpha value of 77 in the water cube colors causing the water to be transparent to some degree.

The second interesting function is the one that updates the point masses used in the demo. For each point mass, the function checks how much of the mass' volume is immersed. Then the point mass is sent to the air and the water environment accordingly to let their forces act on it. Once all forces are set for the point mass, its updating function is called. Basically that is all you need to do in your physics simulations. Since we do not have collision detection yet we need to do another step here. To prevent gravity from sucking the spheres right through the bottom of the room we simply check the height of each point mass against the height of the room's floor. If a point mass starts to penetrate the floor, it's set back to rest onto the floor by brute force.

```
void MySipogenApp::UpdatePointMasses()
{
    float fSubmerged = 0.0f;

    for ( int i=0; i<m_iNumMasses; i++ )
    {
        // check if at least partially inside water
        fSubmerged = m_vMasses[i].GetVolumePercentBelowY(
            m_EnvWater.GetSurfaceY() );

        if ( fSubmerged <= 0.0f )
        {
            m_EnvAir.ActOn( m_vMasses[i] );
        }
        else if ( fSubmerged < 1.0f )
        {
            m_EnvAir.ActOn( m_vMasses[i], 1.0f - fSubmerged );
        }
    }
}
```

```

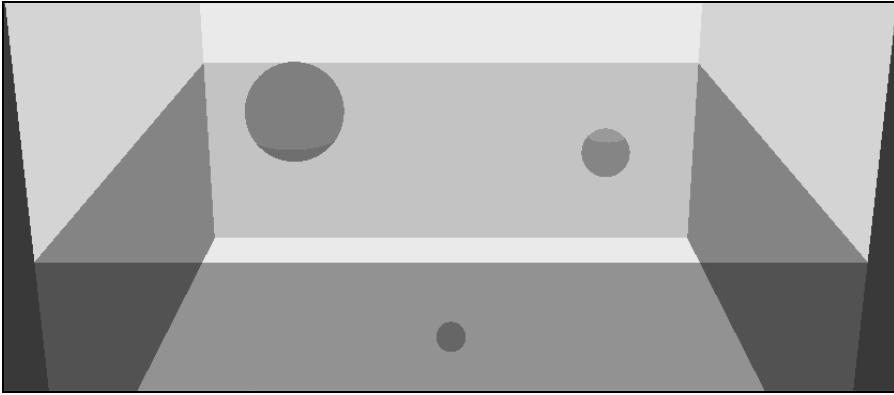
        m_EnvWater.ActOn( m_vMasses[i], fSubmerged );
    }
    else
    {
        m_EnvWater.ActOn( m_vMasses[i] );
    }

    // update all acting forces
    m_vMasses[i].Update( GetElapsedTime() );

    // enforce object to hold at the sea ground since we
    // don't have real collision detection / response yet
    spg::Vector4 vcPos = m_vMasses[i].GetPosition();
    if ( (vcPos.GetY() - m_vMasses[i].GetRadius()) < SEA_FLOOR )
    {
        vcPos.SetY( SEA_FLOOR + m_vMasses[i].GetRadius() );
        m_vMasses[i].SetPosition( vcPos );
        m_vMasses[i].SetLinearVelocity( spg::Vector4(0,0,0) );
    }
}
} // UpdatePointMasses
/*-----*/

```

Figure 9 shows a screenshot from the demo application. There you can see three spheres with different volumes and different masses. The shot is taken after some amount of time. The spheres that originally started falling down from the ceiling have reached a stable position due to their weight and volume. As you can see the left-most sphere has comparably greater buoyancy than the middle sphere, which rests on the room's bottom. The right-most sphere has a comparable gravitational force and buoyant force, but still the buoyancy wins and enables to sphere to have at least a small percentage of its volume lurking out of the water.



*Figure 9: A Screenshot showing the demo application.*

Now you should try out the demo and play around with changing the sphere's mass and radius to see what happens. For those of you who are not yet familiar with the **Sipogen** engine it will also be a good start to look at the rendering code in the demo application. If you are ready to move on, the next chapter will teach you the basics of collision detection.