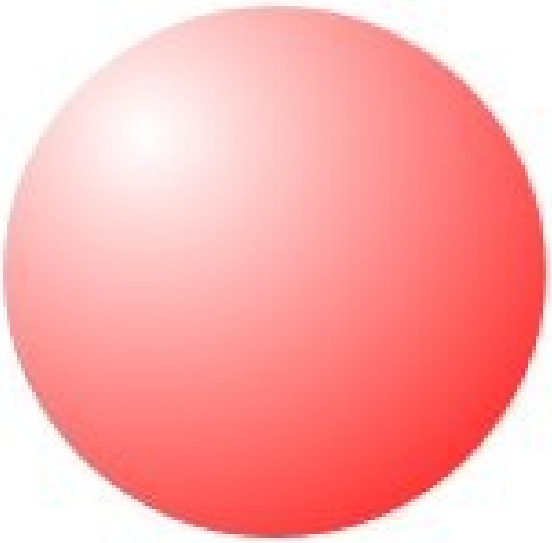
Basic Physics in ActionScript



Introduction:

The first step, anyone should take into learning Flash would be to go through the basic lessons included with the program. You should be familiar already with basic animation and drawing techniques, however, you will not use any of those in these lessons. The key focus in the following lessons will be on code.

The challenge you will face in this tutorial will be to create a functioning program that consists of only one frame. This seems daunting at first, but after you have finished you will be able to extend this style of design to work with animation. We'll get to that in the Afterword.

First thing's first you need to know some of the basic setup techniques.

Movie Clips

Another seemingly unorthodox thing we will do in this lesson is to have no actions in our one lonely frame. Our frame is destined to be that much lonelier. Our code will be instead embedded onto a movie clip.

You can create a **movie clip** by selecting a drawing and either hitting the F8 Key or selecting Modify -> Convert to Symbol.... As you may know movie clips are entities that can have its own **timeline**, but as you may not know, they also may have their own localized variables and are capable of running their own lines of code. After you have created a movie clip you can select an instance of the clip and open the Actions panel.

Clip Events

The foundation for just about everything you will ever do in ActionScript is the **onClipEvent**() handler. Here is the Syntax:

```
onClipEvent(event)
{
    actions;
}
```

Whatever event you select, and there is a prompt that comes up as you type where you can select an event, determines when the actions are ran. If for example you use '**enterFrame**', the actions will be called every frame (30 fps = 30 actions per second). If you were to select '**load**', the actions would be run only once when the clip was loaded. Let's look at an example.

```
onClipEvent(load)
{
	trace("loading...");
	var status="happy";
}
onClipEvent(enterFrame)
{
	trace(status);
}
```

The **trace** command outputs a message to the debug window when you play our movie in the Flash program. The output does not show up in the **swf** files. What trace is useful for is to test movies for certain bugs like trying to find out what a variable's value is at a specific point or seeing how far a piece of code gets before it crashes. In our case, though, we're just using it to have a basic "Hello World" action in our first lesson.

So when our clip loads, it should output "loading..." and create a **local variable** called 'status'. This variable is unique to this movie clip and is at first set to "happy".

Every time our clip enters a frame, it should trace the value of the 'status' variable.

So if you ran this code, the output should resemble;

```
loading...
happy
happy
happy
..etc
```

Actually it will look more like this:

```
happy
happy
happy
happy
happy
happy
happy
happy
happy
happy
```

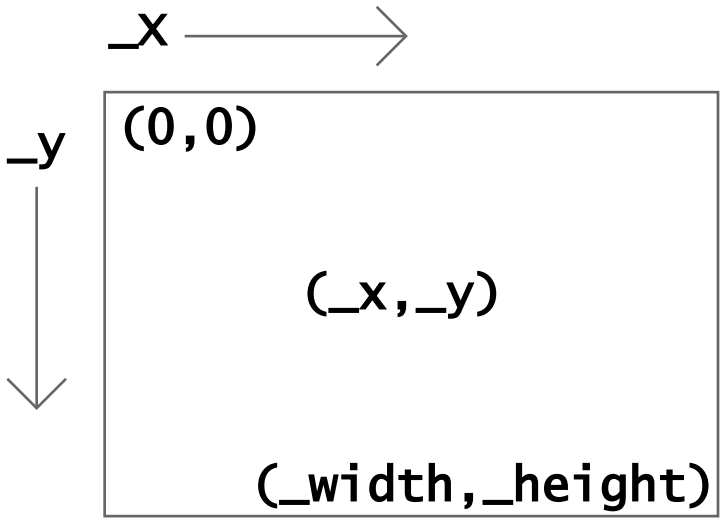
Just running infinitely.... Close that crap if you actually did this part.

Lesson 1: Movie Clips and the Coordinate Plane

OK. Once you've read and understood the prior, more boring section, it is time to get down to business. The basic component of physics programming is the coordinate system. This is just basic x and y middle school geometry.

Every movie clip you create has a wide variety of proprietary variables that you have access to without having to create them. Two of these that we will deal with right now, are **_x** and **_y** (most or all proprietary variables begin with '_', ie: _rotation, _x, _y, _alpha, etc). A movie clip's _x and _y values are initially whatever x and y values the movie clip has in the properties panel, however in ActionScript setting one of these variables to a specific value overrides that, ie: _x=200; _y=150;

The _x and _y values are represented on the stage like this;



So let's dive right in, shall we?

First off create a simple drawing of some sort, it doesn't matter what, and convert it to a movie clip. Then on that movie clip you should put this code.

I recommend against copying it out of this lesson, typing code your self helps you remember it better.

```
onClipEvent(load)
{
	_x=200;
	_y=200;
}
```

Start off with that and and play your movie. Your drawing should automatically move to (200,200). Easy enough, let's continue. Add this next section of code to the same move clip.

```
onClipEvent(enterFrame)
{
	_x=_root._xmouse;
	_y=_root._ymouse;
}
```

Simple again, but this time we've used new variables. **_xmouse** and **_ymouse** are the _x and and _y of the mouse. That's it really they're just the mouse x and mouse y, but _xmouse and _ymouse are global variables therefore you have to precede them with **_root** because they are not local to the movie clip we are working on**.** Simple yes? Run this now.

Your drawing should have followed the mouse around if everything was done correctly, however, you may notice two things that aren't exactly right.

1. The movement is kinda jerky unless you move the mouse slowly.

This is a symptom of a bad habit that almost all beginners in flash have. They tend to use the default of 12 frames per second. I typically use 30. The more frames per second you have the smoother your movie look, this becomes very evident when your working with something real-time like _xmouse and _ymouse. If you use too many frames per second, then the code for enterFrame has to run many more times per second and can cause your movie or game to run slowly. In my practices i find that 30 tends to be optimum for anything that runs code every frame and 60 will even work for things that don't (ie: mouseMove or keyDown). So, from now on try to use a higher frame rate than the default. It just looks better. You can adjust the frame rate in the Modify->Document... window.

2. The drawing is off center.

This is an important thing to learn about movie clips. A movie clips _x and _y are related to the movie clip's **anchor point.** You can see the anchor point by selecting the movie clip. It should look like this:

Granted, you probably don't have a grayish sphere like I do, but the blue box and the tiny cross are the important things. That tiny cross is the anchor point. Obviously your anchor point is what locks onto to the mouse x and y. To change this, you simply double click the movie clip to access the inside of it. Select all of your drawing and move it. Notice that the anchor point doesn't move when you do this. Situate the drawing so that the anchor point is in the what you would like to be the center, and try the program again. Does it center on the mouse now? If it does then you can move on to the next lesson.

Lesson 2: Speed and Reflection

What we learned in the last section, is that every movie clip like all objects in our universe have a position in space. We learned how to relate that to the 2D stage of the flash movie. Objects in space also have a certain speed. If a runner is moving at 5 feet per second, then if we were to observe the runner once every second he should have move 5 feet from the last observation.

We can apply the same principle on a per-frame basis in an enterFrame scenario:

```
onClipEvent(load)
{
	var xspeed=5;
	var yspeed=0;
	_x=50;
	_y=200;
}
onClipEvent(enterFrame)
{
	_x+=xspeed;
	_y+=yspeed;
}
```

Try that. Your drawing should move smoothly across and eventually off the screen. Play with this for a bit changing the values of xspeed and yspeed. Feel free to make them negative or excessively large.

You should have noticed that different values make the object move in a different direction. If neither xspeed or yspeed = 0 then your object will move diagonally. Through manipulation of xspeed and yspeed you can make the object move in any direction.

For now **set xspeed to 5 and set yspeed to -5.** Then add the following code to enterFrame section:

```
if(_x<0 || _x>550)
	xspeed*=-1;
if(_y<0 || _y>400)
	yspeed*=-1;
```

The whole '||' thing is an OR statement. "if y is less than 0 OR y is greater than 400".

yspeed*=-1; is a basic reflection. Multiplying a speed by -1 reverses it. Like bouncing off of a wall.

When you run this, it should simply bounce around the screen. Once again try different xspeeds and yspeeds and experiment. You can also experment with the boundaries. I used 0-550 and 0-400. That's the default width and height of a new movie.

Now for a little player control. The next bit of code uses a lot of new things, so I'll go over it. **Add this somewhere after the enterFrame section.**

```
onClipEvent(keyDown)
{
	if(Key.isDown(Key.LEFT))
		xspeed=-5;
	else if(Key.isDown(Key.RIGHT))
		xspeed=5;
	if(Key.isDown(Key.UP))
		yspeed=-5;
	else if(Key.isDown(Key.DOWN))
		yspeed=5;
}
```

I assume you already know how to use **if** and **else if.** Another type of ClipEvent is keyDown. These actions run every time a key is pressed. That's any key. if you put actions in there with no if statement the code would run every time you pressed anything. So if you use if statements regarding which key is being pressed you can assign actions to a specific key.

The basic syntax is if(Key.isDown("key")). "key" can be any integer value based on the ascii key code. The a key is represented by 97 b is 98 and so on. Using Key.LEFT, however simplifies things by using special predefined key variables. Really, Key.LEFT = 1007 or something like that.

So experiment with this code by setting different starting coordinates and different x and ypseeds for the keys to assign.

Now you should decide which you would rather work on; Trajectory or Acceleration and Gravity? Lessons 3 - 5 are all sequential taking you through basic to advanced trajectory methods, while lessons 6 - 9 all deal with acceleration and the subtle manipulation of speed to emulate gravity.

I highly recommend both subjects.

Lesson 3: Basic Trajectory

First things first, **make a little dot somewhere on the screen. Select that dot and go to the properties panel and change its x and y to 200.** Here is the code for your movie clip.

```
onClipEvent(load)
{
	_x=200;
	_y=200;
	var xspeed=0;
	var dx;
	var yspeed=0;
	var dy;
}
onClipEvent(mouseDown)
{
	_x=200;
	_y=200;
	dx=_root._xmouse-200;
	dy=_root._ymouse-200;
	if(Math.abs(dx)>=Math.abs(dy))
	{
		xspeed = (dx/Math.abs(dx))*10;
		yspeed = (dy/Math.abs(dx))*10;
	}else {
		xspeed = (dx/Math.abs(dy))*10;
		yspeed = (dy/Math.abs(dy))*10;
	}
}
onClipEvent(enterFrame)
{
	_x+=xspeed;
	_y+=yspeed;
}
```

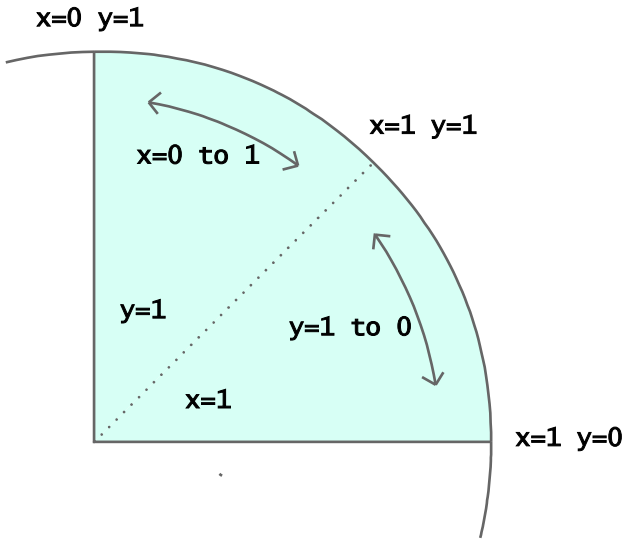
First off, mouseDown is just like keyDown, in that it occurs whenever you click the mouse. The variables dx and dy are the distance of _xmouse from the dot and the distance of _ymouse from the dot. Math.abs(var) is the absolute value of var. A really simple formula for figuring out an xspeed and yspeed based on mouse position is this:

xspeed = dx divided by absolute value of larger distance multiplied by the maximum speed

yspeed = dy divided by absolute value of larger distance multiplied by the maximum speed

So basically, to determine an objects x and y speed based on trajectory you have to find a way such as the previous forumla to produce a value between 0 and 1. You then take that value and multiply it by the maximum speed. So if your maximum speed is 10, you end up with some value between 0 and 10.

With this formula you get an imperfect, but workable result. Here's 1 quadrant:



This formula is easy to use but difficult to understand. Don't worry, though, in Advanced Trajectory you will learn a much more sound principle.

SO! **Try out this very nifty little piece of code if you would. Experiment with the max speed** because it should be the only variable that will affect this game.

Lesson 4: Rotation and Trigonometry

Don't let the header of this section scare you. Trigonometry does make the average person sick to their stomach and I'm no exception. In fact when I was first learning the following principle I actually couldn't remember a single thing from high school trig. However, by learning to implement trigonometry in Flash, I now have a working understanding of it. Hopefully by the time this is over you will too.

The first thing we will do is cover a few of the finer points of the code for this section.

_rotation = an objects rotation in degrees with a default of 0. So if a movie clip is right side up, meaning you haven't rotated it any by using the transform tool, that will be what it looks like at zero degrees _rotation. At 180 degrees rotation it will be upside down. You get the idea.

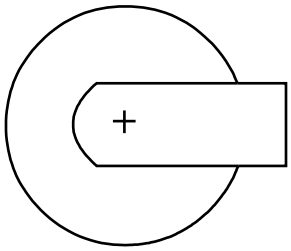
Math Object = a set of proprietary values and functions that will allow you to do complex calculations. For example, Math.PI = 3.14 or some other approximation of PI.

These include but are not limited to:

function	*syntax*	*usage*
power	Math.pow(value,power)	Math.pow(x,2)
square root	Math.sqrt(value)	Math.sqrt(x)
rounding	Math.round(value)	Math.round(3.467892)
random number between 0 and 1	Math.random() * range	Math.random()*range
Sin,Cos,etc	Math.sin(radians)	Math.sin(_rotation*Math.PI/180)

So let's get started with some practical application. The first thing we will do is make a kind of gun turret that will rotate to point at the mouse.

You should first draw a circle and center it on the anchor point like we did in an earlier lesson, then you should add some kind of barrel, just make sure it points to the right. You should end up with something similar to this:



You need to make sure that the barrel is obvious and points to the right, and that the anchor point is not in the center of the entire drawing but only of the circle.

```
onClipEvent(load)
{
	var adjside;
	var oppside;
	var angle;
	var MAXSPEED=20;
}
onClipEvent (mouseMove) {
	adjside = _root._xmouse-_x;
	oppside = -1*(_root._ymouse-_y);
	angle = Math.atan2(oppside,adjside);
	// in radians
	angle = Math.round(angle/Math.PI*180);
	// convert to degrees
	_rotation = -1*(angle);
}
```

Place the preceding code on the new clip and test it out. The turret should follow the mouse exactly. Try double clicking the movie clip in the Flash editor to go inside to the drawing, and then add a long red line extending out past the turret like a laser. The line should always touch or come very close to the mouse.

The formula for the angle an object should face to point to a certain coordinate in space is this:

angle in radians = arctangent of y/x when y = the vertical distance between the origin and the object and x = the horizontal distance between the origin and object

That gives us the rotation but in radians. To convert it to degrees, you simply use this formula:

angle in degrees = angle in radians / Pi * 180

So now we have a turret that follows the mouse. Why don't we shoot something out of it? **Keep this project open and move on to the next lesson.**

Lesson 5: Advanced Trajectory

So far we have a turret that follows the mouse, and a ball that will shoot towards the mouse. It would be simple work to simply combine those two and have a turret that shoots a ball at the mouse, but I promised a better formula for the ball thing earlier.

The formula we used earlier should only be used if you are just wanting to determine a trajectory based on x and y position, but since we already have a turret that figures out the rotation to the mouse, why not use rotation to determine our ball's trajectory?

Here is the forumla:

xspeed = Cos(angle*Math.PI/180);

yspeed = Sin(angle*Math.PI/180);

That's it. That is all the code you need to determine the ball's x and y speed. **So create a ball. It doesn't have to be complicated just a small circle in a color you can see easily. Convert it to a movie clip. Center it up like we've done before. Now, click on it, go to the properties tab, and where it says** <Instance Name>**, name the object ball.**

Go back to the turret and add the following code:

```
onClipEvent(mouseDown)
{
	_root.ball._x=_x;
	_root.ball._y=_y;
	_root.ball.xspeed=Math.cos(_rotation*Math.PI/180)*MAXSPEED;
	_root.ball.yspeed=Math.sin(_rotation*Math.PI/180)*MAXSPEED;
}
```

So when you click the mouse, it tells _root.ball (your ball) that its x and y should be equal to the turrets x and y. That starts the ball off at the center point. Then it runs our xspeed and yspeed formula and sets the ball off on its course.

Speaking of the ball, click on it and add our basic xspeed yspeed code:

```
onClipEvent(enterFrame)
{
	_x+=xspeed;
	_y+=yspeed;
}
```

Experiment with this, play with it. If you want, you could add more code to the ball to make it bounce around the screen with using the code from Lesson 2.

There are innumerable ways you could implement this into a game scenario. You could have a ship that was flying around and turrets that aimed at it and shot lasers. You could have a tank that rolled around slowly and had a turret on top that targeted the mouse and allowed you to fire shells at enemies.

In fact there is a very popular game made by PopCap Games called Zuma, that uses this very logic.

You can check it out at Popcap.com

Lesson 6: Acceleration

Before we get started on the basics of acceleration, it would probably be a good idea to review basic speeds and the KeyDown system.

Let's begin by simply making another ball. Make this one fairly large, if you want, and convert it to a movie clip.

Begin by adding this code to the...

You know what, actually this would be a good time to practice. **Add to the movie clip code to make it move right up and down at a certain speed using the arrow keys, but use enterFrame instead of keyDown.** I'll explain later.

Hint: Right = _x + =speed; Left= _x - =speed;

What you should end up with is basic movement control like that you would find in any RPG or adventure game. We are gonna do something a little more complex with it though.

Replace your code with this:

```
onClipEvent(load)
{
	var xspeed=0;
	var yspeed=0;
	var MAXSPEED=15;
}
onClipEvent(enterFrame)
{
	if(Key.isDown(Key.LEFT) && xspeed>MAXSPEED*-1)
		xspeed--;
	else if(Key.isDown(Key.RIGHT) && xspeed<MAXSPEED)
		xspeed++;
	if(Key.isDown(Key.UP) && yspeed>MAXSPEED*-1)
		yspeed--;
	else if(Key.isDown(Key.DOWN) && yspeed<MAXSPEED)
		yspeed++;
	_x+=xspeed;
	_y+=yspeed;
}
```

Notice the difference in this and the code you replaced it with. Instead of having the ball instantly move at a certain speed, it gradually speeds up. This is a much more realistic approach to movement, however it's very situational and should not be used all the time. Take for example, a flying game. Top down, lateral movement shoot the enemies like Raiden or Galaga or even Centipede. None of those games would be very fun if you had to dodge something and couldn't immediately get moving. Games like that require quick response, but some games, especially ones that involve balls or vehicles, could do for a slow acceleration, and gradual slow down. **Speaking of slowing down, add this code to the enterFrame section:**

```
xspeed *= 0.9;
yspeed *= 0.9;
```

Now every time the clip enters a frame, it loses 10% of its speed. Try it out and see for your self. You might have noticed that 0.9 slows the ball down *a lot*, well it does. 10% is way too much when you do it around 30 times a second. Try changing the value to make it more realistic. Maybe have it go down by 5%.

Once everything is working, try adding the code from previous sections to make it bounce when it hits a boundary.

What you have here is the potential for an addictive Marble Madness or Labyrinth type game where you need to have some touch in moving the marble around and not let it get away from you. Some of you ..... some of you game nerds could even I dunno make this into a .... *cough*Katamary Damacy clone*cough*hint*cough.

Lesson 7: Dynamic Acceleration

IMPORTANT* This lesson is a huge step from the other lessons! Its nothing you can't handle but it is somewhat difficult to explain, I will try my best. If you want to do this lesson then I HIGHLY recommend that you go back and read Lesson 5: Advanced Trajectory, and to do that you should probably read Lessons 3 and 4 as well. You have been warned!

First things first, 'dynamic' is one of those funny buzz that many people throw around. Sadly, not many really understand the definition in the context of what they're talking about. In this case, dynamic acceleration means that we are going to have an object accelerate based on its direction. So xspeed and yspeed are going to equal 'dynamic' values depending on what direction the car is facing. Don't worry there is a rather simple formula for all of this, understanding what the formula does is the hard part.

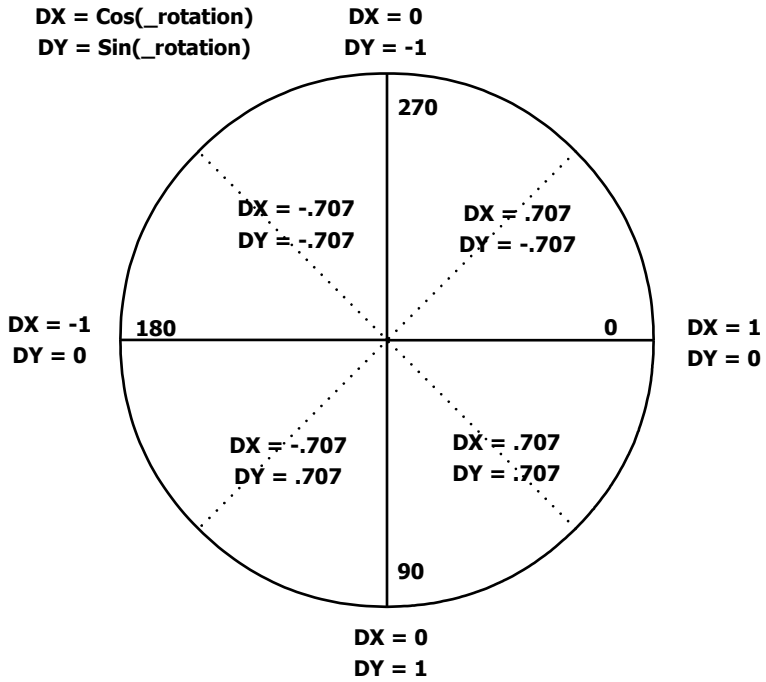
Since we'll be working with a car shaped block, we will call it **Formula 1:**

speed = some number between 0 and max speed depending on acceleration.

xspeed = Cos(_rotation * PI / 180) * speed;

yspeed = Sin(_rotation * PI / 180) * speed * -1;

There that wasn't so bad. So! You have a single measurement of speed that you change when you accelerate. When you hit the gas, if you will. That speed, or let's be more scientific and say magnitude, is then coupled with the object's rotation and given an x and y speed. Just in case you didn't read Lesson 5, that is where the trig functions come in.



I realize none of this is adding up to a car game, so let's go ahead and get rolling, so to speak. **Draw a basic car.** It has to be from a bird's eye view, and I would prefer it be very simplistic. At its most 'pimped out' it should look like this:

Add this code to it:

```
onClipEvent(load)
{
	var speed=0;
	var xspeed;
	var yspeed;
	var SPEEDLIMIT = 20;
}
onClipEvent (enterFrame) {
	if (Key.isDown(Key.UP)) {
		speed += 1;
	}
	if (Key.isDown(Key.DOWN)) {
		speed -= 1;
	}
	if (Math.abs(speed)>SPEEDLIMIT) {
		speed *= .7;
	}
	if (Key.isDown(Key.LEFT)) {
		_rotation -= 15;
	}
	if (Key.isDown(Key.RIGHT)) {
		_rotation += 15;
	}
	speed *= .98;
	xspeed = Math.sin(_rotation*(Math.PI/180))*speed;
	yspeed = Math.cos(_rotation*(Math.PI/180))*speed*-1;
	_x+=xspeed;
	_y+=yspeed;
}
```

We've done a whole lot of different things in this code, the most of basic of which is the steering. When you hit the left and right keys it simply changes the rotation of the car. Right increases it by 15 degrees left does the opposite. The up key works as a gas pedal increasing the speed of the car, while the down key works not so much as a brake but more of a reverse accelerator (it will make the car go in reverse if you hold it down).

Every frame, the car calculates its x and y speed using Formula 1, moves based on the x and y speed, and slows the overall speed down by 2% unless the cars speed is over the speed limit in which case it slows the car by 30% essentially bringing it under the limit.

Try that out. Did you notice how neatly the car turned though? Turns on a dime without losing any speed. Very unrealistic. A real car should lose some speed when turning. **So add this code into the Key.LEFT and RIGHT sections.**

```
speed*=0.95;
```

That should make the car decelerate properly while turning, but realize that 0.95 is just a number that I've tested and looks right to me. You should always test these things out for your selves. **So try experimenting with a few of the numbers in this code.** You can change any value you want to get different results, but avoid messing with Formula 1.

Lesson 8: Basic Collision

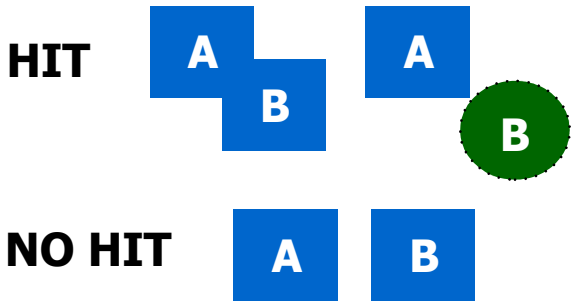
This will be a bit of a break away from acceleration but you need it for the next lesson. Let's discuss the hitTest method.

Syntax: targetA.hitTest(targetB)

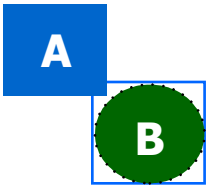
Usage: if(needle.hitTest(balloon))

```
{
        //KABOOM!
}
```

Demonstration:



The reason that Block A and Sphere B are colliding is because all objects have a bounding box. Even though the green B is a circle it is surrounded by a box like this:



Therefore, A is colliding with B when it really shouldn't be. This is a big problem down the road, especially if you're working a large slope or something, but its not that hard to get around. Also, when you're dealing with a small squarish object like this it really doesn't make much difference.

So recreate something from a past section that moves on your command. Preferably not the car that would be a little too complicated. I would recommend the object from Lesson 6. Name the object ball in the properties panel.

Now draw a large thin block. Make it into a vertical wall and then a movie clip and add the following code :

```
onClipEvent(enterFrame)
{
        if(this.hitTest(_root.ball))
        {
                _root.ball.xspeed*=-1;
        }
}
```

Try that out. The ball should bounce off of the wall (excluding the bottom and top of it). That's nice and all if you're doing a marble or billiards kind of thing, but something more practical you can do (if you ever wanna make platform or adventure games) is to make it stop the object.

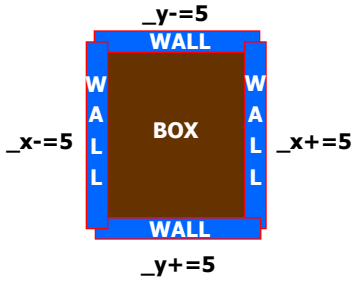
Remember the code you created in Lesson 6 to make an object walk left and right (_x+=5 or _x-=5)? **Dust that code off and make a new object with it.** You should have an object that just moves in the direction you press.

Replace the code on the wall with this:

```
onClipEvent(enterFrame)
{
	if(this.hitTest(_root.ball))
	{
		_root.ball._x-=5;
	}
}
```

'5' is of course whatever you used in your code. If you used _x+=10 to make your's move right then use 10 instead. This code will only work if you approach the wall from the left moving right, but if you did everything correctly, you should have a wall that doesn't let you pass.

That only covers a right side collision though, so if you were doing this in a real adventure type game you would set your walls up like this:



Bring back the accelerating ball from that we used at the start of this lesson.

Replace the wall's code with this:

```
onClipEvent(enterFrame)
{
	if(this.hitTest(_root.ball))
	{
		_root.ball._x-=_root.ball.xspeed;
		_root.ball.xspeed=0;
	}
}
```

That makes the ball hit the wall and stop. It opposes the ball's force then sets the ball's speed to 0, making it collide and not pass and stopping it at the same time. We will use that method of collision physics in the next section.

Lesson 9: Gravity

This is the culmination of this entire chapter of tutorials. If you are into making platform games, you will use this code over and over and over again so pay attention.

First, draw a stick man. Doesn't have to be Michaelangelo's David or anything, just make sure his anchor point is at his feet. **Name him guy.**

Give your guy this code:

```
onClipEvent(load)
{
	var gravity=0;
	var TermVeloc=30;
}
onClipEvent(enterFrame)
{
	if(gravity<TermVeloc)
		gravity++;
	_y+=gravity;
}
```

Make another wall, this time horizontal. Give it the following code:

```
onClipEvent(enterFrame)
{
	if(this.hitTest(_root.guy))
	{
		_root.guy._y=_y-1;
		_root.guy.gravity=0;
	}
}
```

Try that out. Your guy should fall until he comes to rest on the platform. Experiment by putting him different heights above the platform. He should always come to rest on the platform.

Essentially, in every frame, the guy's gravity increases until it gets to the terminal velocity (max speed). When he hits the platform, it tells him that his feet should be just barely above the platform (_root.guy._y=_y-1;) and that his gravity should return to 0.

Now let's add control to the enterFrame section on the guy:

```
if (Key.isDown(Key.LEFT)) {
        _x -= 10;
} else if (Key.isDown(Key.RIGHT)) {
        _x += 10;
}
if (Key.isDown(Key.SPACE)) {
        _y-=5;
        gravity = -10;
}
```

The left and right are just your basic movement methods, but the space is much more interesting. First, it lifts the object up just a little to move it out of hitTest range from the floor. Then it sets its gravity to -10. When that happens the object moves upward until its gravity eventually levels it out and drops it. So basically the space bar makes your guy jump. Try that out. Play with it have fun with it.