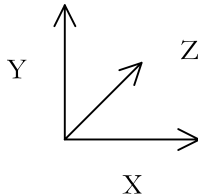
3D Rotation Tutorial

This is a tutorial on how to rotate objects in 3D in Flash. Actually, "revolve" or "orbit" are probably better words. The objects themselves won't rotate, but will move around a central point.

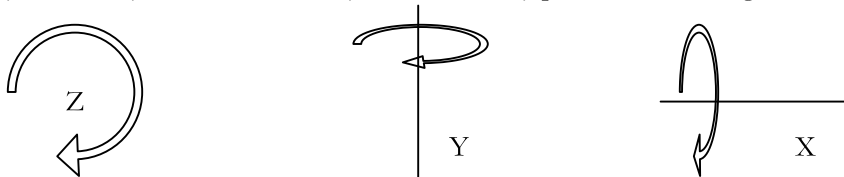
First, a primer on 3D. Three dimensions. Width – left to right, height – top to bottom, and depth – close to far. A monitor screen can show you only two dimensions – width and height. To show depth, we have to resort to various tricks, the most obvious being that as things get further away, they get smaller. Also, objects will tend to move toward a central "vanishing point" as they move away from you, and will tend to fade out to a degree depending on the atmosphere. All these effects can be created relatively easily in Flash.



On computers, height is generally represented by Y, or in Flash, the _y, _yscale and _height properties. Width is X or the _x, _xscale and _width properties. Depth is represented by Z. Since Flash is a 2D program, it doesn't have a _z property, but we can create one.

Next, let's talk about rotation. There are three ways to rotate (or orbit) an object.

Z-rotation rotates something around the z-axis. Think of a clock, or a steering wheel. The object doesn't get any closer to you, or further away, but its x and y position will change.



Y-rotation is around the y-axis. Think of a merry-go-round or a record player. The height (y) doesn't change, but x (left to right) and z (depth) do change.

X-rotation around the x-axis as you might have guessed. Think of a blade on a circular saw. The height (y) and depth (z) change, but it doesn't move left to right (x).

Flash supports z-rotation for movieclips through the _rotation property. But this is just straight rotation – it will spin around its own center. If we want to orbit the object around a central point, like a planet around the sun, we need to get more advanced.

Since z-rotation is the easiest, we'll start out with that, even though it's not really 3D just yet.

The basic problem is how to move something around a center point. It will always be the same distance from that center point, say 100 pixels. So we have a radius of 100, and an angle which will go from 0 to 360. From this data, we need to calculate the _x and _y positions. Here is the formula:

```
_x=Math.cos(angle)*radius+xcenter;
_y=Math.sin(angle)*radius+ycenter;
```

I'm not going to go into an explanation of the trigonometry involved here. It's actually pretty simple, but I'll leave that for another tutorial. Just know that this works. A very important point you need to know is that "angle" needs to be in radians. This is a different system from the 360 degrees that you might be used to. Since it's a lot easier to think in degrees, you'll need to convert them. The formula for that is:

```
rad=degree*Math.PI/180;
```

Now you just need a loop that increases the angle from 0 to 360 and you will have circular motion. To pull it all together, make a movieclip of some object and put the following code on it:

```
onClipEvent(load){
	speed=5;
	radius=100;
	xcenter=250;
	ycenter=200;
	angle=0;
}

onClipEvent(enterFrame){
	_x=Math.cos(angle*Math.PI/180)*radius+xcenter;
	_y=Math.sin(angle*Math.PI/180)*radius+ycenter;
	angle+=speed;
	if(angle>359){
		angle-=360;
	}
}
```

Test that and you should have your mc moving around in a circle. Not 3D yet, but an important step along the way.

OK. Assuming you got that working, let's try y-rotation. Imagine the circle you just had, but you pull the top toward you and push the bottom away until it is laying down flat. Its "Y" is now its "Z" but "X" remains the same. So all we have to do us substitute z for _y:

```
onClipEvent(load){
	y=0;
	speed=5;
	radius=100;
	xcenter=250;
	ycenter=200;
	zcenter=100;
	angle=0;
}

onClipEvent(enterFrame){
	_x=Math.cos(angle*Math.PI/180)*radius+xcenter;
	_y=y+ycenter;
	z=Math.sin(angle*Math.PI/180)*radius+zcenter;
	angle+=speed;
	if(angle>359){
		angle-=360;
	}
```

```
}
```

Notice I also added a zcenter value in the load section and changed ycenter to zcenter in the enterFrame section. You can try playing around with this value. A higher number will make the whole animation happen further away from you. A lower one will obviously make it come closer. I also computed a value for _y. Otherwise _y would be 0 and the whole thing would be on the top edge of the screen.

If you test this now, the object will just move left to right. Although we have computed its z value, we haven't done anything with it. We will now use it to generate some scaling so we have a bit of a 3D effect.

So, how much to scale it to get a realistic perspective? Now we need to get into the field of optics, believe it or not.

First we need to set a focal length, which is the theoretical distance from the viewpoint to the screen. How much you make this value will determine how much perspective things have – how fast they will grow or shrink as they move toward or away from you. We'll set it at 150 for now. You can fool around with it later to create the exact effect you want. The focal length isn't going to change, so we'll set it in the "load" section of the code.

```
fl=150;
```

Next we need a scale value. This will be based on the focal length and z. The formula is:

```
scale=fl/(fl+z);
```

You can work out the physics of this if you want, but it ends up with a scale as a decimal between 0 and 1. When z=0, scale will be 1.0. As z goes into the distance (increases toward infinity), scale will decrease downward toward 0. We can then use this to compute an _xscale and _yscale for our object:

```
_xscale= _yscale = scale*100;
```

Our final code:

```
onClipEvent(load){
	y=0;
	speed=5;
	radius=100;
	xcenter=250;
	ycenter=200;
	zcenter=100;
	angle=0;
	fl=150;
}

onClipEvent(enterFrame){
	_x=Math.cos(angle*Math.PI/180)*radius+xcenter;
	_y=y+ycenter;
	z=Math.sin(angle*Math.PI/180)*radius+zcenter;
	scale=fl/(fl+z);
	_xscale= _yscale = scale*100;
	angle+=speed;
	if(angle>359){
```

```
            angle-=360;
        }
}
```

Test that. We're looking pretty good so far. But we can make it even more realistic.

I also mentioned earlier that as an object retreats into the distance, it moves in toward the vanishing point. In our example above, we have y=0, which is eye level and already at the vanishing point, so you wouldn't see a change. But say we put y=100. It would now be below our eye level and we would expect to see it move upwards as it went into the distance. Similarly, if it were less than 0, it would be above us and it should move down toward the horizon as it goes away.

The same holds true for the x value. As the object moves away, it should move toward the center of our vision horizontally. You don't notice it as much here, because it's already moving left and right. But let's be perfectionists.

For _x, we will first calculate the position on the circle, assigning that to "x", then adjust for perspective and assign that to _x. Here:

```
onClipEvent(load){
        y=100;
        speed=5;
        radius=100;
        xcenter=250;
        ycenter=200;
        zcenter=100;
        angle=0;
        fl=150;
}

onClipEvent(enterFrame){
        z=Math.sin(angle*Math.PI/180)*radius+zcenter;
        scale=fl/(fl+z);
        x=Math.cos(angle*Math.PI/180)*radius;
        _x=x*scale+xcenter;
        _y=y*scale+ycenter;
        _xscale= _yscale = scale*100;
        angle+=speed;
        if(angle>359){
            angle-=360;
        }
}
```

Note that we need to compute z and scale first, so it is available to the rest of the code. Try changing the initial y value to get the object rotating at different heights. Try negative numbers too.

That's about as realistic as you can get in terms of scaling and positioning. For the finishing touch, you can add some atmosphere by computing an _alpha based on scale. Just throw in:

```
_alpha=scale*100;
```

You can add more or less atmosphere by playing around with that 100 number.

The next logical step is to add multiple objects moving around in a circle. To have them evenly spaced, you just need to adjust the initial angle of each one. Divide 360 by the number of objects and add that to each successive object.

For example:
You have 10 objects. 360/10 = 36. Initial angle for each object:

Object 1: angle =36;
Object 2: angle =72;
Object 3: angle =108;
etc…

You can do this by hand, or automatically with a for ( ) loop.

An additional note:
I coded this with onClipEvent's for ease of explanation. I would recommend putting all this code inside of the clip, on its own time line. i.e. set all the constants (the stuff in onClipEvent(load))in frame one of the clip, perform all the actions (stuff in onClipEvent(enterFrame)) in frame 2, and have frame 3 gotoAndPlay(2); That way, the code is built into the object and you can just drag it out of the library to make more copies, use it in another movie, etc. without having to recreate all the onClipEvent actions.

By the way, if you now want to make some x-rotation, just go through the above code and replace every reference to a y-type value with x, and every x with y.