Bodies and Constraints

Author: Kristof Aldenderfer

In the last section, you used Particles - coupled with simple forces - to create a satellite orbiting a planet. While elegant in its simplicity, this structure has limitations. What if you want to rotate the Particles themselves? How do you constrain them to certain parts of the screen? Particles have no vocabulary for expressing rotational dynamics, because they are zero-dimensional objects; forces cannot bind objects to dimensional limits. You'll need utilize **bodies** and **constraints** to really unleash the expressive power of the physics engine.

This section covers how to:

- create more expressive physics objects using bodies
- use angular momentum when building bodies to make them rotate
- use torque to apply rotational forces to bodies
- bind physics objects to dimensional limits

Body Properties

Physics is used to describe the motions and interactions of objects, and since you exist in three physical dimensions, so should your famo.us physics objects! The **Body** is the multidimensional equivalent of the Particle, and there are two types to choose from: the **Rectangle** and the **Circle**. They inherit all properties of particles (**mass**, linear **velocity**, and the ability to be acted on by the forces already covered). In addition, you can specify dimensional values: **size** for rectangles, and **radius** for circles.

Build bodies just as you would particles, and include a dimensional property:

```
var circle = new Circle({
    radius: [70],
    mass: 16,
    position: [0, 80, 0]
});

var xRectangle = new Rectangle({
    size: [50, 50],
    mass: 1,
    position: [-120, 0, 0]
});
```

Angular Velocity and Angular Momentum

Angular velocity describes how fast an object is spinning about its own axis. This is the rotational equivalent to linear velocity; just as a sprinter has a speed and a direction (eg. 10meters/sec south), a rotating object has a rotational speed and direction (eg. 2rad/sec clockwise around the z-axis).

Angular momentum describes rotational motion, but takes into account the body's *angular velocity, mass,* and the *geometry*. This is the rotational equivalent of linear momentum. A cyclist and a bus may

have the same velocity, but vastly different momenta because of their disparity in both mass and geometry. In the same way, while the sun and the moon actually have very similar angular velocities, they have vastly different angular momenta.

Both angular velocity and angular momentum can be set either during creation of the body, or on the fly afterward using the appropriate set method. They both take a three-dimensional Vector. For angular velocity:

```
var xRectangle = new Rectangle({
    size: [50, 50],
    mass: 1,
    position: [-120, 0, 0],
    angularVelocity: [0.2, 0, 0]
});
```

or (after creation):

```
xRectangle.setAngularVelocity([0.2, 0, 0]);
```

For angular momentum:

```
var yRectangle = new Rectangle({
    size: [100, 100],
    mass: 1,
    position: [0, 0, 0],
    angularMomentum: [0, 0.5, 0]
});
```

or (after creation):

```
yRectangle.setAngularMomentum([0, 0.5, 0]);
```

Example: Angular Momentum

Each rectangle is given the same total angular momentum. Even though the red and green rectangles have the same mass, because the red is half the size of the green, it spins faster. The red and blue rectangles are the same size, but because the blue is more massive, it spins slower. Note the following lines of code:

context.setPerspective(*value*) determines how the view is projected onto the screen. A larger *value* creates a perspective which is “farther away”; that is to say, rotations and translations in space are less pronounced. A smaller *value* creates very pronounced rotations and translations. Try swapping *value* between 1000 and 10 to see its effects.

backfaceVisibility: 'visible' sets the rear side of Surfaces to be seen. By default, if you rotate a Surface anywhere between 90° and 270°in either the x- or y-axis, it will be invisible!

Rotational Forces

Bodies respond to forces, both linear and rotational. A force which causes a body to rotate is called a **torque**. Just as the force applied to an object determines its change in velocity, a torque applied to an object determines its change in angular velocity. Applying the RotationalDrag force to a spinning body causes it to slow down and eventually stop:

```
var rotationalDrag = new RotationalDrag({
    strength: 0.01
});

physics.attach(rotationalDrag, xRectangle);
```

But flipping the sign on its strength will cause it to spin faster.

Since RotationalDrag is attached via the physics engine, it is a constant force acting on the body. For one-time and triggered rotational forces, bodies contain the *torque* property. It can be applied during creation:

```
var zRectangle = new Rectangle({
    size: [100, 100],
    mass: 1,
    position: [120, 0, 0],
    torque: [0, 0, 0.6]
});
```

or as an instantaneous rotational force later on:

```
zRectangle.applyTorque([new Vector(0, 0, 0.8));
```

Contrast the language used for the torque method (“apply”) with that of angular velocity and momentum (“set”). Torque is an action taken on an object – not an inherent physical measurement of the object itself, as angular velocity and angular momentum are. Torque is a “verb”, the others “nouns.” This also means that torque is additive: successively applying torque creates a cumulative effect on the body. Imagine pushing the bars of a children's merry-go-round as they rotate by you. Each push ultimately adds more angular velocity.

Example: Rotational Forces

Each rectangle is given an initial angular momentum, this time during creation. Now, however, rotational drag is attached to each, which eventually slows them to a stop. The smaller the mass and dimensions of the object, the more quickly it will come to a stop.

An on-click event will apply a torque to the rectangles. Multiple clicks make the rectangles spin even faster, because torque is an action, and therefore additive.

Constraining a Body with Walls

One of the consequences you'll notice about setting velocities of and applying forces to bodies is that inevitably one of your objects will zoom off the screen, never to return. You may eventually also want to limit the distance between two bodies. Luckily, the physics engine has an answer for both of those scenarios! Built into famo.us are **constraints** which bind physics objects to dimensional requirements. They're the precursors to full body-to-body collisions, and they attach to objects in the same way as forces. The simplest is the **Wall**. Walls have a **normal** property, which determines which direction it is facing, and a **distance** property which determines how far away from the origin it is placed. To make a floor on your screen for a circle to be reflected off of:

```
var floor = new Wall({
    normal : [0,1,0],
    distance : window.innerHeight/2
});
```

In this instance, the *normal* property says to point in the positive Y direction: up! All that is left is to attach the wall to the circle:

```
physics.attach(floor, circle);
```

To create a container in which to trap a circle, you can create four walls – but the resulting code is fairly repetitive. To save you time and coding space, you can use the **Walls** constraint:

```
var walls = new Walls({
    size : [500, 500],
    align: [0.5, 0.5]
});

for (var wall in walls.components) {
    physics.attach(walls.components[wall], circle);
}
```

Example: Walls Constraint

Capture that circle inside a box! Giving it an initial velocity allows you to see the collisions.

More Complicated Constraints

Several other constraints are available; the **Snap** constraint binds an object to a specific position (or another body) using the **anchor** property. This constraint acts very similarly to a spring force, except that the response is much quicker. The **period** sets the length of time it takes the body to return back to the anchor position when it is displaced, and the **dampingRatio** determines how quickly the oscillations subside. The higher the ratio, the more the snap constraint is damped, and therefore the fewer the oscillations.

The compliment to the snap constraint is the **Distance** constraint, which binds a body to remain a

certain distance away from a position (or another body). In addition to setting the anchor, period, and dampingRatio, you can set the **length** of the constraint, which defines the actual distance from the anchor it will bind to.

Example: Snap and Distance Constraints

Combining the Snap and Distance constraints, you can create a wave-like effect on a group of circles when one is clicked. Each circle snaps to a specific positional anchor on the screen: `ball.home`. Each circle is *also* distance-constrained to its neighbors: `ball.rope`.

When one circle is displaced in the -z direction with a click, `ball.rope` pulls its neighbors with it, while `ball.home` pulls it back to its original position. Adding slight displacement in the -y direction exaggerates the effect.

The function `makeBall` is where all the object creation is done. Each ball contains a surface, a circle, a home, and a rope property. The ball is then added to an array of balls, which is then added to both the physics engine and to the main context in the `balls.forEach` method. It is here that the constraints are attached as well.

Change the `message` variable to see how the variable `dist` changes the alignment of the elements.