P5 Box2D Physics Tutorial

Foreword: The tutorial is the Physics chapter from **Introduction to Programming with JavaScript, P5 and Processing** by Robert Cook on amazon.com. Please send any corrections or suggestions to kindlecbook@gmail.com. Thank you for your purchase. The code in the tutorial can be reused under the M.I.T. license.

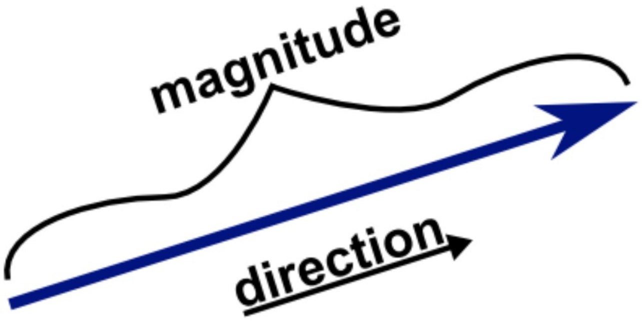
"**Processing** is an open source programming language and integrated development environment (IDE) built for the electronic arts, new media art, and visual design communities with the purpose of teaching the fundamentals of computer programming in a visual context, and to serve as the foundation for electronic sketchbooks." P5 is the pure JavaScript implementation of Processing.

Skip to the Physics Section if you already understand Vectors and basic trigonometry.

Vector Class

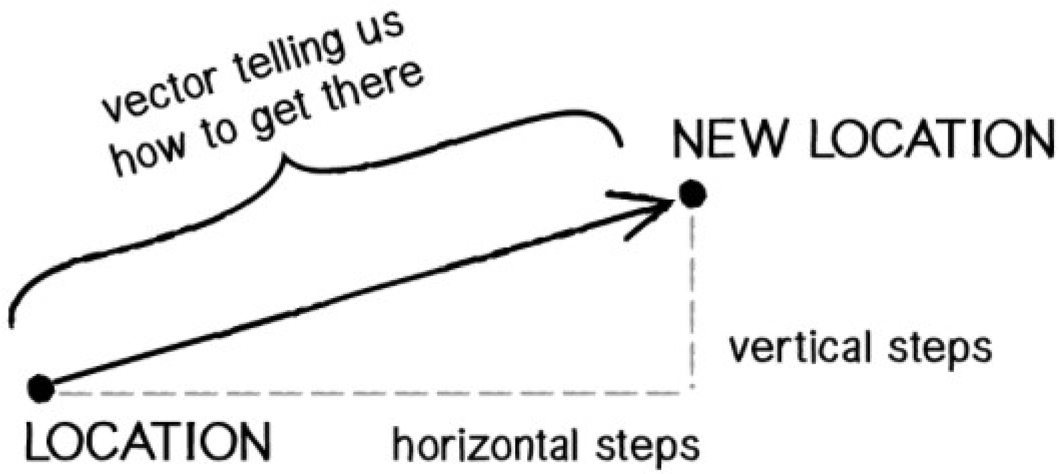
One of the basic building blocks for programming motion is the vector, which is named for the Greek mathematician Euclid and is also known as a geometric vector. A vector defines a magnitude (or length) and a direction in a Cartesian coordinate system graph. A vector is usually drawn as an arrow. The magnitude of the vector is its length while its tip indicates the direction.

A Vector



Obviously, a line requires a point at each end; however, the length of a line and its direction can be represented by translating the line's starting point to the origin (0,0). As a result, a vector can be described by its end point only (a single xy pair of values). In physics, **velocity** is a vector that can be applied to a point to change its location.

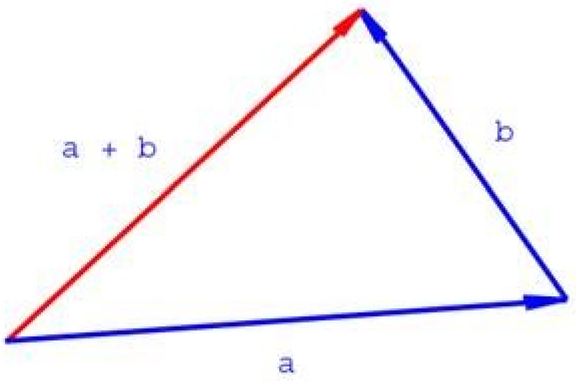
Velocity Vector



Another term from physics is **acceleration**, which is the addition of a vector to the velocity of a point. The Earth's standard acceleration due to gravity is g = 9.80665 m/s^2 (32.1740 ft/s^2). Other physical forces (wind, lift, drag, friction, buoyancy, magnetism etc.) can also be coded as vectors.

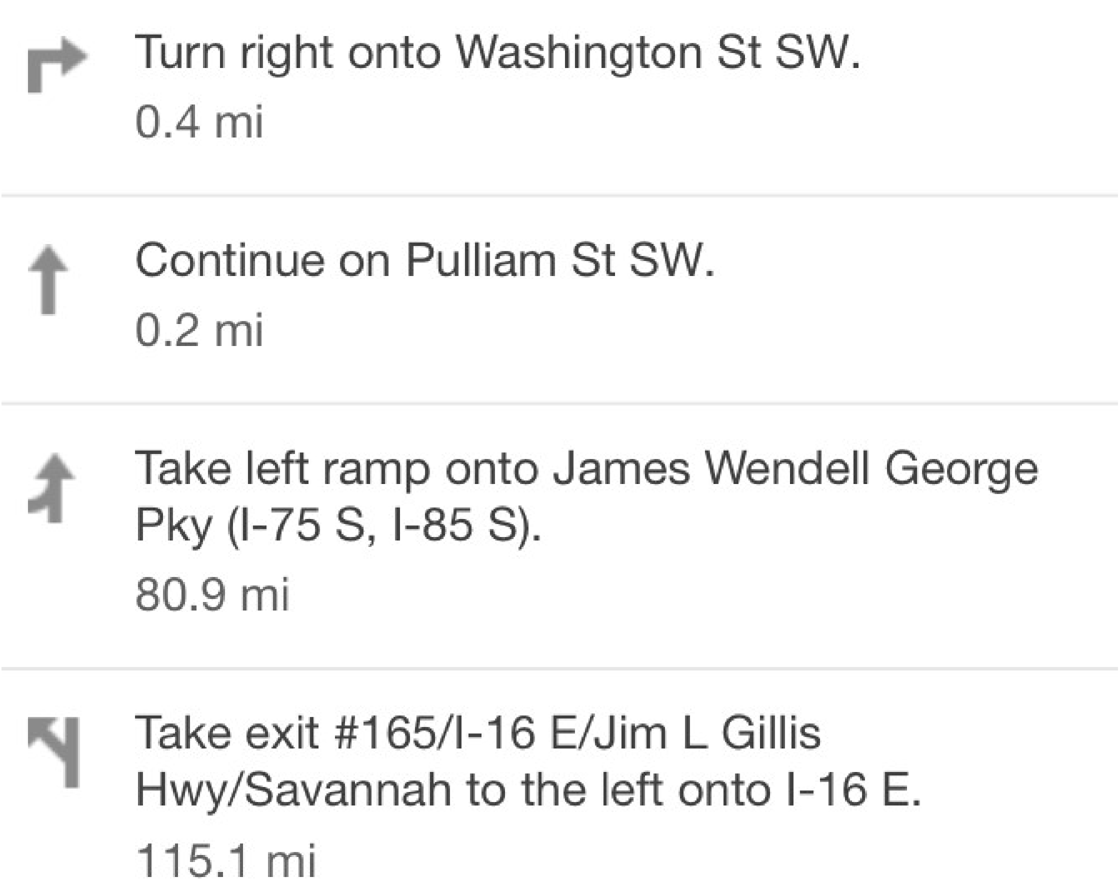
An operator, such as vector addition, is just the application of a change in location. For example, go to the refrigerator in the real world might translate into 6 steps east then 9 steps north in the vector world. Similarly, vector subtraction would simply reverse the path from the refrigerator so that we ended up where we started. In the Figure, (a+b)-b is the vector a. Vector multiplication scales a vector by an x-direction multiple and a y-direction multiple.

Vector a+b



A common human use of vectors is a list of driving directions from one city to another. Each segment is a direction/magnitude vector. How does the program calculate the distance?

Driving Directions



Turn right onto Washington St SW.
0.4 mi

Continue on Pulliam St SW.
0.2 mi

Take left ramp onto James Wendell George Pky (I-75 S, I-85 S).
80.9 mi

Take exit #165/I-16 E/Jim L Gillis Hwy/Savannah to the left onto I-16 E.
115.1 mi

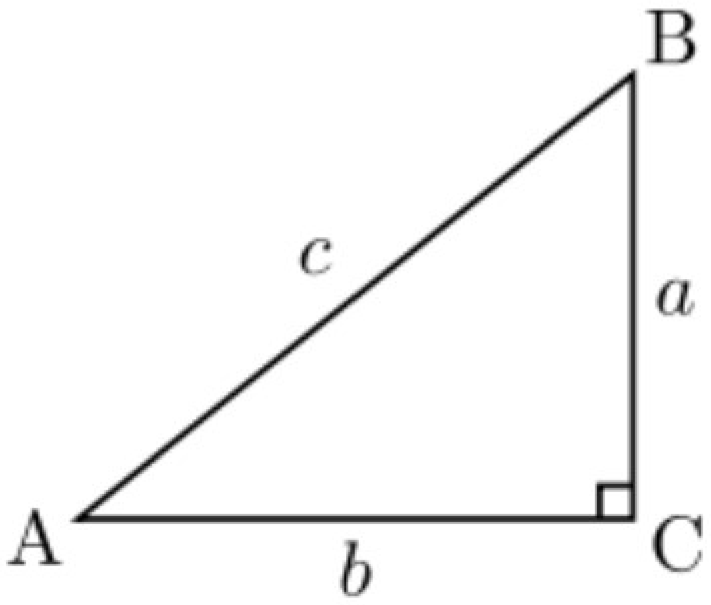
The **length**, or **magnitude**, of a vector is calculated using geometric principles that have existed for thousands of years. Even cave people had to guess how tall a tree was so it would not crush anything when felled.

A circle represents a 360 degree arc that can be divided into four 90 degree quadrants (X Y, -X Y, X -Y, -X -Y) on a Cartesian graph. In most mathematical work, angles are typically measured in **radians** rather than degrees.

A right triangle is a three-sided figure in which one angle is a **right angle** (that is, a 90-degree angle). The relation between the sides and angles of a right triangle is the basis for **trigonometry**.

The side opposite the right angle is called the **hypotenuse** (side c in the figure). The small square at the right angle is the typical symbol used to denote that an angle is 90°.

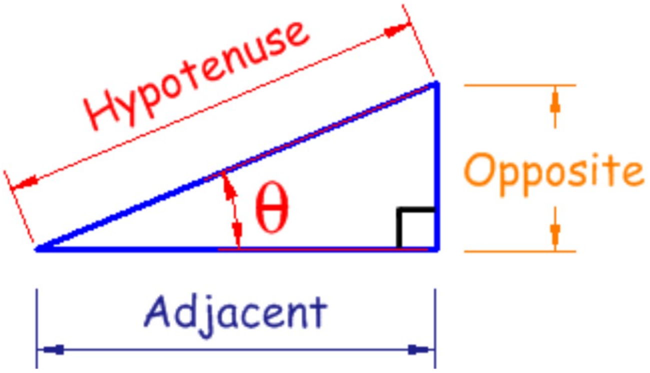
Right Triangle



In the Figure, (b,a) is the vector notation for c. We wish to calculate the length of c. The **Pythagorean Theorem**—or Pythagoras' theorem—is a relation in geometry among the three sides of a right triangle. It states that the square of the length of the hypotenuse (the side opposite the right angle) is equal to the sum of the squares of the lengths of the other two sides. The theorem can be written as an equation relating the lengths of the sides a, b and c, often called the Pythagorean equation: $a^2 + b^2 = c^2$. The Pythagorean theorem is named after the Greek mathematician Pythagoras (ca. 570 BC—ca. 495 BC), who by tradition is credited with its proof. The length of a vector (b, a) is calculated as sqrt($a^2 + b^2$).

Observe that in the right triangle in the Figure, there are two angles that are unknown. Typical problems that have challenged humans for thousands of years are to be given lengths and to be asked for angles or to be given an angle and asked for lengths. Thus, the science of **trigonometry** was born. Three of the main functions in trigonometry are **sine**, **cosine** and **tangent**. They are all based on analysis of a right triangle.

Right Triangle Side Names



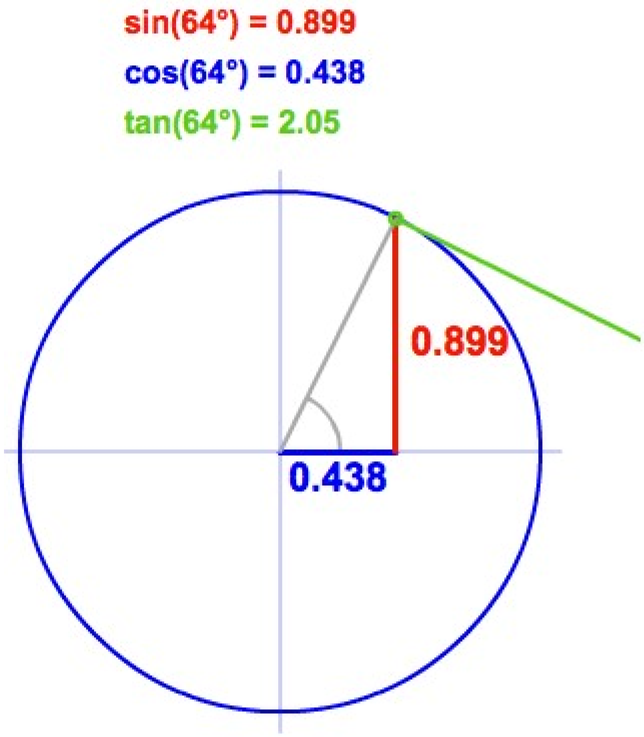
Remember that there are two angles of interest. Either angle can be calculated based on a ratio of side lengths. The **hypotenuse** is always the side opposite the right angle and is the longest side. The **adjacent** side forms the angle of interest together with the hypotenuse. The **opposite** side is the line connecting the hypotenuse to the adjacent side. The ratios are calculated as listed in the table. The mnemonic to help you remember the definitions is **soh cah toa.** Given the side lengths, the two angles can be computed; given the angles, the two side lengths can be computed.

Function	Ratio
sine(angle in radians)	Opposite / Hypotenuse
cosine(angle in radians)	Adjacent / Hypotenuse
tangent(angle in radians)	Opposite / Adjacent
tangent(angle in radians)	sine(angle) / cosine(angle)
angle in radians	arc cos(Adjacent / Hypotenuse)

There are an infinite number of vectors that have the same direction, but different lengths (1,1) (2,2) (0.8,0.8) etc. It would be useful if we could represent all of the vectors that point in the same direction as just one vector. In math, this goal is termed a **canonical representation** or a **normal form**. For vectors, the canonical representation of all vectors with the same direction is called a **unit vector**. It should be intuitive that if all the lengths are different but the direction is the same, the unit representation for a vector can be calculated by dividing by its length, which results in a length of one.

In the sine/cosine table, if the hypotenuse's length is one, the sine and cosine give the length of the Opposite and Adjacent sides directly. Observe that a unit vector has a length of one. As a result, the distance from the hypotenuse to the X or Y axis can be calculated. The relationship between a unit vector and side lengths is frequently depicted as a unit circle, which is shown next. The length of the Opposite side is 0.899/1 and the length of the adjacent side is 0.438/1 for an angle of 64 degrees.

Unit Vector



The JavaScript math library includes the described trigonometric functions in addition to the inverse functions, such as Math.acos(), which will invert a cosine result to its argument angle (in radians).

Another useful geometric concept is the **dot product**, which is used in graphics as well as in other fields, such as mechanics. The dot product (v1 · v2) of two vectors is the cosine of the angle between them, multiplied by the lengths of the vectors, v1 and v2. So, you can easily calculate the cosine of the angle by either making sure that the two vectors are both of length one, or by dividing the dot product by the lengths. The actual angle between the two vectors can be determined by applying Math.acos to the cosine result.

Angle Between Two Vectors

DotProduct(v1,v2) = cosine(angle) * length(v1) * length(v2)
cosine(angle) = DotProduct(v1,v2) / (length(v1) * length(v2))

It has been proven (see Wikipedia) that the geometric definition of the dot product is equivalent to the algebraic definition. Since the algebraic definition is more efficient to calculate, it is the one implemented in the Vector class.

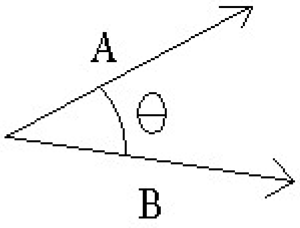
```
function dot(v1, v2) {
  return v1.x*v2.x + v1.y*v2.y;
}
```

dot((.438,0), (.438,.899)) = adjacent dot hypotenuse = (.438*.438 + 0*.899) / (.438 * 1) = .438
Math.acos(.438) = 1.117 radians
degrees(1.117) = 63.999 degrees

The dot product values range from 1 to -1. If the two input vectors are pointing in the same direction, then the return value will be 1. If the two input vectors are pointing in opposite directions, then the return value will be -1. If the two input vectors are at right angles, then the return value will be 0. To solve for angle theta, use the geometric dot product formula. The Math.acos() function can be used to calculate the angle between two vectors by dividing the dot product of two vectors by their lengths.

Dot Product

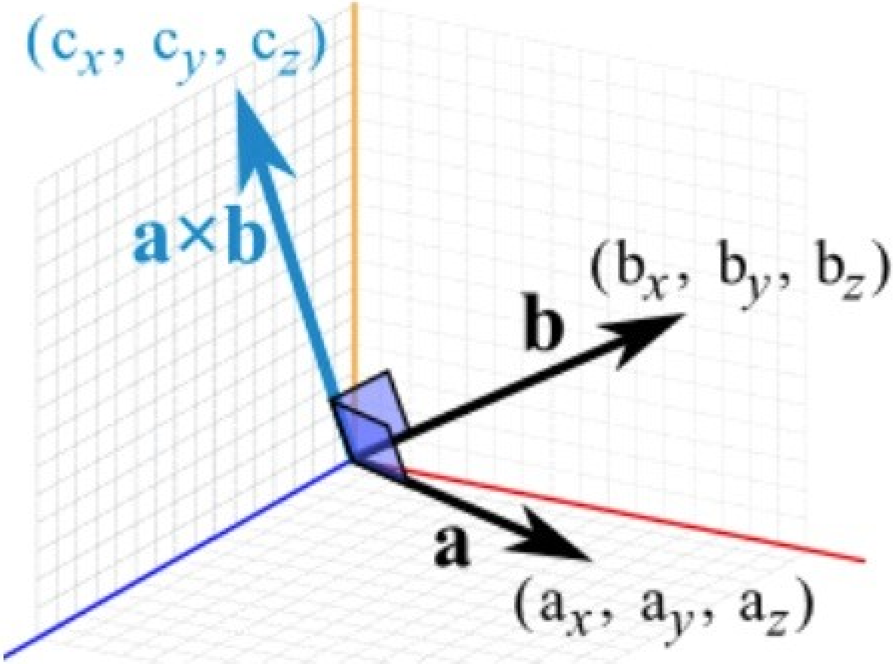
$$A \cdot B = |A|\,|B|\,\cos\theta$$



The cross product is another geometric concept to discuss before proceeding to the design of the Vector class. The **cross product** of two vectors (v1 x v2) is another vector that is at right angles to both. It has uses in three dimensions that we will not be discussing.

We are interested in the cross product in two dimensions because its magnitude c_z can be interpreted as the area of a parallelogram having side lengths of only v1 and v2. Given two unit vectors, their cross product has a magnitude of 1 if the two are perpendicular and a magnitude of zero if the two are parallel.

Cross Product

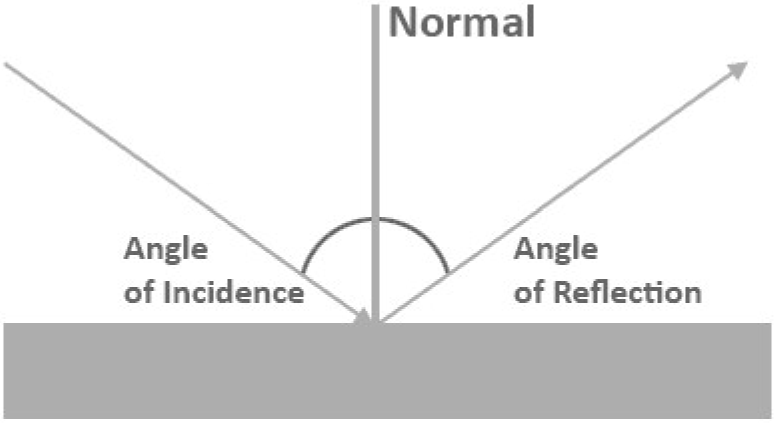


$c_x = a_y b_z - a_z b_y$
$c_y = a_z b_x - a_x b_z$
$c_z = a_x b_y - a_y b_x$

```
function cross(v1,v2)
  return v1.x*v2.y - v1.y*v2.x
}
```

In physics and in gaming, lots of things (balls, light, cars, bullets) bounce off of other things. There is a law for that which states that the **angle of reflection** is equal to the **angle of incidence** with respect to a surface's normal vector. Basically, a ball bounces off a surface at the same angle at which it arrived. The **normal** of a flat surface is a vector that is perpendicular to it. Note that a "normal" vector is a different concept than normalizing a vector.

Angle of Reflection



The P5 library implements a vector class, which includes all of the methods discussed. The following table summarizes the design of the Vector class.

P5 Vector Methods

createVector(x,y [,z])	Creates a new p5.Vector instance.
new p5.Vector(x,y [,z])	Creates a new p5.Vector instance.
v.**add**(x,y) v.**add**(v2)	Adds x, y, and z components to a vector, adds one vector to another, or adds two independent vectors together. The version of the method that adds two vectors together is a static method (p5.Vector.add) and returns a p5.Vector, the others act directly on the vector.
p5.Vector.**angleBetween**(v,v2)	Calculates and returns the angle (in radians) between two vectors.
v.**array**()	Return a representation of this vector as a float array.
v.**copy**()	Gets a copy of the vector, returns a p5.Vector object.
v.**cross**(v2)	Calculates and returns a vector composed of the cross product between two vectors. Both the static and non static methods return a new p5.Vector.
v.**dist**(v2)	Calculates the Euclidean distance between two points (considering a point as a vector object).
v.**div**(scale)	Divide the vector by a scalar. The static version of this method creates a new p5.Vector while the non static version acts on the vector directly.
v.**dot**(v2)	Calculates the dot product of two vectors. The version of the method that computes the dot product of two independent vectors is a static method.
v.**equals**(v2)	Equality check against a p5.Vector.
p5.Vector.**fromAngle**(radians)	Make a new 2D unit vector from an angle.
v.**heading**()	Calculate the angle of rotation for this vector (only 2D vectors).
v.**lerp**(v2, percent)	Linear interpolate the vector to another vector.
v.**limit**(max)	Limit the magnitude of this vector to the value used for the **max** parameter.
v.**mag**()	Calculates the magnitude (length) of the vector and returns the result as a float (this is simply the equation sqrt(x*x + y*y + z*z)).
v.**magSq**()	Calculates the squared magnitude of the vector and returns the result as a float (this is simply the equation _(x*x + y*y + z*z))_. Faster if the real length is not required in the case of comparing vector lengths, etc.
v.**mult**(s)	Multiply the vector by a scalar. The static version of this method creates a new p5.Vector while the non static version acts on the vector directly.
v.**normalize**()	Normalize the vector to length 1 (make it a unit vector).
p5.Vector.**random2D**()	Make a new 2D unit vector from a random angle.
p5.Vector.**random3D**()	Make a new random 3D unit vector.
v.**rotate**(radians)	Rotate the vector by an angle (only 2D vectors), magnitude remains the same.
v.**set**(x,y, [,z])	Sets the x, y, and z component of the vector using two or three separate variables, the data from a p5.Vector, or the values from a float array.

v.**setMag**(len)	Set the magnitude of this vector to the value used for the **len** parameter.
v.**sub**(x,y) v.**sub**(v2)	Subtracts x, y, and z components from a vector, subtracts one vector from another, or subtracts two independent vectors. The version of the method that subtracts two vectors is a static method and returns a p5.Vector, the other acts directly on the vector.

Examples

http://jsfiddle.net/bobcook/5n0nbr9y

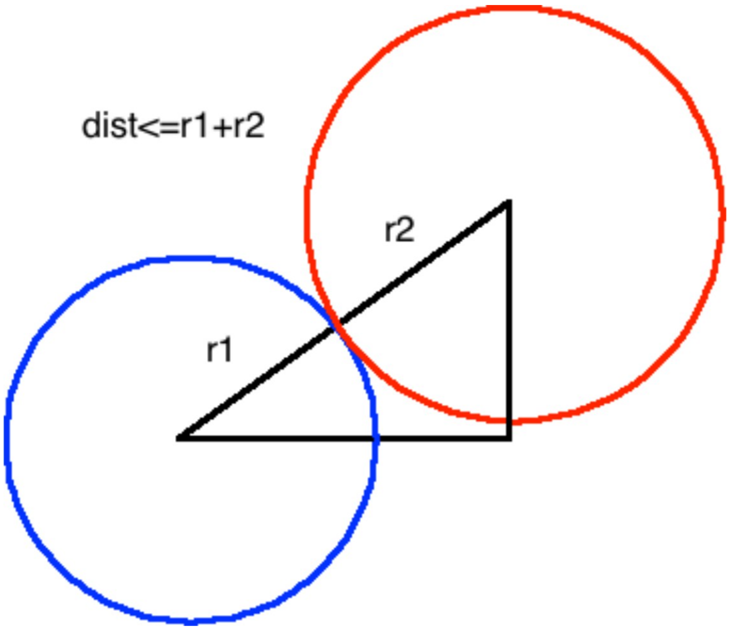
Code	**Output**
function setup() {	
createCanvas(600, 400);	
var v = new p5.Vector(4,5);	
out(v);	v=4 5
out(v.add(1,2));	v+1,2 = 5 7
out(v.add(v));	v+v = 10 14
out(p5.Vector.add(v,v));	v+v = 20 28
var y = createVector(0,4);	y = 0 4
var x = createVector(10,0);	x = 10 0
var r=p5.Vector.angleBetween(x,y);	
out(degrees(r));	angle(x,y) = 90
out(v.array().length);	v.array.length = 3
var r = createVector(0,0);	r = 0 0
//r=x.copy();	
r.y = 9;	
out(x);	x = 10 0
out(r);	r = 0 9
out(x.cross(y));	cross(x,y) = 0 0
out(x.dist(y));	dist(x,y) = 10.770329614269007
out(v.div(2));	v/2 = 5 7
out(x.dot(y));	dot(x,y) = 0
//out(x.equals(y));	
out(p5.Vector.fromAngle(PI/2));	frmAngle = 6.123233995736766e-17 1
out(x.heading());	heading(x) = 0
out(r.lerp(3, 3, 0, 0.5));	lerp = 1.5 6
out(r);	r = 1.5 6
out(x);	x = 10 0
out(y);	y = 0 4
out(x.lerp(x,y,0.5));	lerp = 10 0
out(x);	x = 10 0
out(x.limit(2));	limit(x) = 2 0
out(x.mag());	mag(x) = 2
out(x.magSq());	magSq(x) = 4
out(x.mult(2));	x*2 = 4 0
out(x.normalize());	norm(x) = 1 0
out(p5.Vector.random2D());	rand = -0.9983284 -0.0577962

out(p5.Vector.random3D().z);	rand.z = 0.146488015
out(x.rotate(PI/3));	rotate(x) = 0.5 0.866025
out(x.set(3,4));	x = 3 4
out(x.setMag(6));	setMag(x) = 3.599 4.8
out(x.sub(1,1));	x-1,1 = 2.599 3.8
out(x.sub(y));	x-y = 2.599 -0.199
}	

Vector Demonstration

One of the traditional demonstrations for the utility of a Vector class is moving object collision. The sample code implements circle collision because the intersection test is trivial. Two circles intersect if the distance between the centers is less than the sum of the radii.

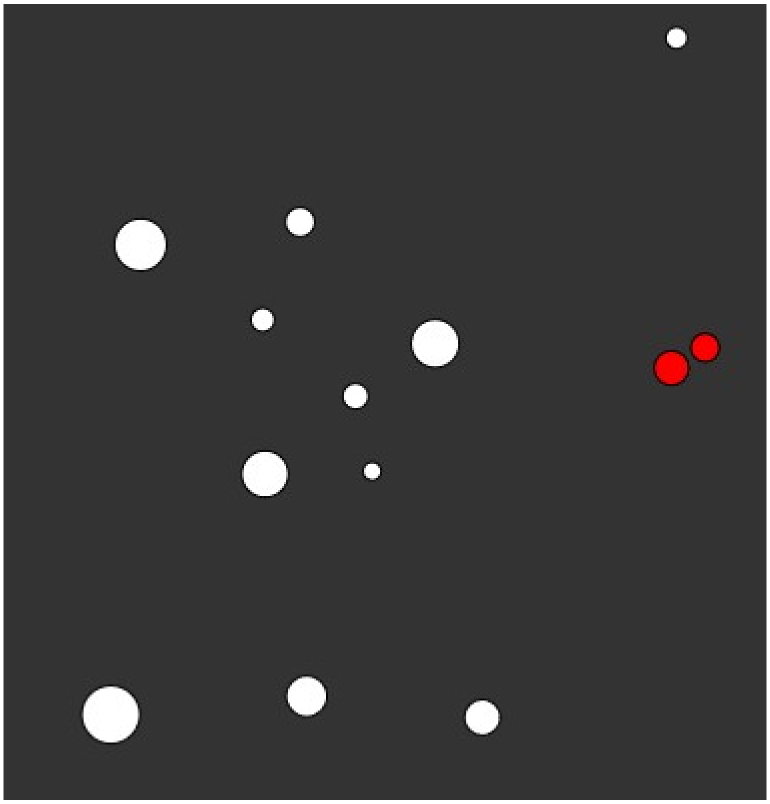
Circle Collision



The main code generates N instances of the Circle class. One complication was that the collision response did not handle circles that were created on top of each other. As a result, the generator code in "setup" had to delete a Circle if it overlapped another and try again.

To add interest, colliding balls turn red and we have added a sound (beep MP3 file). At each "step", every ball must check for a wall collision. Also, each ball must check to see if it is colliding with any other balls. This an expensive test, that at its simplest, requires N balls to check N-1 other balls. This is N^2 operations. There are much better algorithms for collision testing. We opt for simplicity over speed. Also, collision testing gets more complicated for irregular figures.

Bouncing Circles
http://jsfiddle.net/bobcook/mghedLy9/ (run as index.html)



1. The first step was to modify the Circle class to store the xy position as a vector. Next, the "setup" method was modified to generate Circles at random positions. The location range is inset by the largest radius so that no Circle overlaps an edge of the canvas. We also added a "velocity" property to each Circle, but did not use it immediately as step-wise refinement dictated that the new vector/drawing code be tested first.

   The "sound" library can be downloaded from the P5 web site, which also has numerous usage examples.

   ```
   var balls = [];
   var N = 13;
   var beep;
   function preload() {
     beep=loadSound("assets/beep.mp3");
   }
   function setup() {
       createCanvas(400, 400);
       start: for (var i = 1; i <= N; i++) {
           var c = new Circle(
           random(15, width - 15),
           random(15, height - 15),
           random(4, 15));
           c.velocity = p5.Vector.random2D().mult(4);
           for (var j = 0; j < balls.length; j++) {
               if (c.isIntersecting(balls[j])) continue start;
           }
   ```

```
    balls.push(c);
  } //for i
} //setup
```

2. The next step was to loop through the "balls" array incrementing the position by the velocity.

```
for (var i = 0; i < balls.length; i++) {
    balls[i].xy.add( balls[i].velocity );
}
```

3. Then the xy position of each ball had to be tested for intersection with a canvas edge. By testing x and y collisions separately, the corner-collision test is included for free. If a collision occurs, a beep sound is played. Herein occurred the first bug. It turns out that sounds are queued so that if five collisions are detected in a second, the beeps will still be playing long after the circles have gone their separate ways. The solution was to only play a new beep if an old one had stopped playing.

```
function draw() {
  background(51);
  var xy, beeper = false;
  for (var i = 0; i < balls.length; i++) {
    balls[i].xy.add( balls[i].velocity );
  }
  for (var i = 0; i < balls.length; i++) {
    var xy=balls[i].xy;
    var collision = false;
    if (xy.x < balls[i].r || xy.x > (width - balls[i].r)) {
      collision = true;
      balls[i].velocity.x = -balls[i].velocity.x;
    }
    if (xy.y < balls[i].r || xy.y > (height - balls[i].r)) {
      collision = true;
      balls[i].velocity.y = -balls[i].velocity.y;
    }
    collision = collision || balls[i].testAndBounce(balls);
    if (collision) {
      beeper = true;
      balls[i].xy.add(balls[i].velocity);
      fill(255, 0, 0);
    } else {
      balls[i].xy = xy;
      fill(255);
    }
    ellipse(balls[i].xy.x, balls[i].xy.y, balls[i].r*2, balls[i].r*2);
  } //for i
  if (beeper && !beep.isPlaying()) beep.play();
}
```

4. The final step was to implement the "testAndBounce" method, which checks for a ball collision against an array of balls. If a collision is detected, the velocity is modified in the opposite direction proportional to the strength of the collision. For example, a head-on crash is more powerful than a

glancing blow by two circles heading in the same direction.

```
Circle.prototype.testAndBounce = function(circles) {
  for (var i=0; i<circles.length; i++) {
    var b = circles[i];  //reference, not copy
    if (b == this) continue;
    if (b.xy.dist(this.xy) > (this.r+b.r)) continue;
    var dif = p5.Vector.sub(this.xy, b.xy);
    dif.normalize();
    var dot = dif.dot(this.velocity)* -2;
    dif.mult(dot);
    this.velocity.add(dif);
    return true;
  } //for i
  return false;
}
```

PHYSICS FOR FUN

Introduction

Hopefully the vector math and geometry examples that have been introduced so far were sufficient to convince you that traditional "stuffy" subjects can enable cool programs in the world of JavaScript. In this tutorial we do not introduce new programming concepts, rather we introduce you to game physics simply because it is fun to play with!

Box2D Physics Library

The Box2D physics engine (box2d.org), which was originally written in platform-independent C++, has also been ported to JavaScript. The version (box2d-html5) that is used in the examples is the same as that used in the Nature of Code examples by Daniel Shiffman.

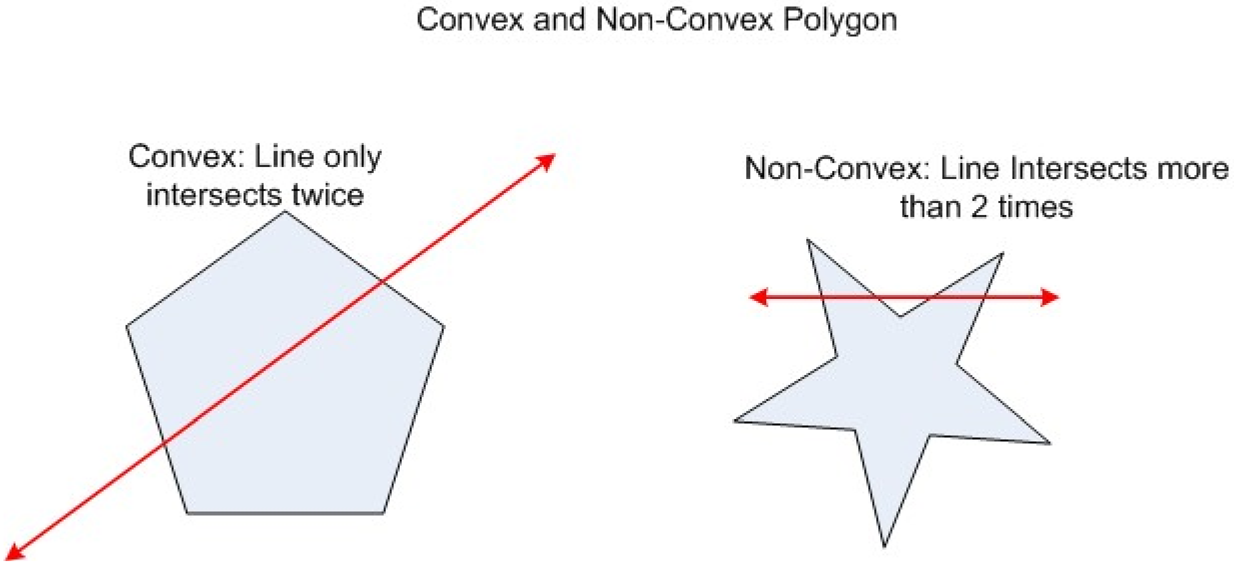
https://github.com/shiffman/The-Nature-of-Code-Examples-p5.js/tree/master/chp05_libraries

Box2D was first released as a demonstration engine to accompany a physics presentation given by Erin Catto at the 2006 Game Developers Conference. The code was posted as an open-source release in 2007.

Box2D implements a constrained, rigid-body simulation that can model bodies composed of edges, rectangles, convex polygons (no concave angles) or circle shapes. Rigid bodies never deform; their perimeter always remains exactly the same no matter what forces are applied. A single polygon is **RESTRICTED** to a maximum of eight vertices.

Bodies are connected together with joints and acted upon by forces. Concave polygons can be implemented by joining multiple convex pieces. The shapes are used only in the physics simulation; they define the boundary of sprites and static barriers, such as buildings, roads, or cliffs. When a program draws a body, the figure has no restrictions. It can be an image or an animation. Remember that the physics shapes are chosen to be a close approximation to the boundary of what is drawn. For debugging, the physics library includes a default drawing routine for all body types.

Box2D was designed to utilize the standard physics units of meters/kilograms/seconds. The simulation algorithms are optimized for object sizes ranging from 0.1m (a grape) to 10 meters (a school bus). The API provides a scale function (up to about 30) to go larger, but not smaller, which translates from about 3 pixels to 300.


Convex and Non-Convex Polygon
Convex: Line only intersects twice
Non-Convex: Line Intersects more than 2 times

The engine also integrates gravity, density, friction, and bouncing (termed **restitution**) calculations as well as collision detection. Gravity is an acceleration vector so objects can fall up, down, sideways, or not all all. Density determines the mass of a body (mass equals density*area). Friction regulates two bodies that are in sliding contact.

The Box2D user controls the physics engine by creating objects with numerical properties. The engine then evaluates a complicated series of formulas that approximate the objects' behavior in the physical world. Be aware that this is an imperfect system. Sometimes the results can be very strange. More often, the results are impressive. Be prepared to experiment with the parameters to achieve your goals. Remember you are using physics, not learning the details from a physics textbook. People drive cars, but not everyone can replace brake linings.

Physics Bodies

Remember that acceleration is the change in an object's velocity over time, but how is acceleration calculated? Sir Isaac Newton defined the following formula in 1687. It is known as **Newton's Second Law of Motion**.

$$\text{Acceleration} = (\sum \text{Forces}) / \text{Mass}$$

One definition of mass is a measure of an object's resistance to acceleration. The standard unit for mass is the kilogram (kg). **The mass of an object is calculated as the density times the area.** The standard unit for force is the newton [N = kg m/s^2], which is a vector quantity that represents mass times acceleration. Notice that dividing force by mass (kg) yields m/s^2, which is acceleration. Distance is measured in meters.

Following the step-wise refinement principle, we explore the features of Box2D physics through a series of small examples. Since this is a book for beginners, we have implemented a "b2" class to facilitate physics coding. The easiest way to get started is to "fork" an existing Fiddle. The URLs that I used for development are listed next. The two JavaScript files can also be downloaded to run local tests via index.html, which enables easier use of the JavaScript debugger.

B2 and Box2d Library Links
https://raw.githubusercontent.com/bobcgausa/cook-js/master/b2.js
https://raw.githubusercontent.com/bobcgausa/cook-js/master/box2d-html5.js

I started work on the B2 library to provide a simple, and understandable, interface to Box2d, which has had the same API since its creation. I never imagined that I would end up with a 20K file comprised of 600 lines of code. There are many tutorials on JavaScript Box2D. If you wish, examine those for comparison.

The first step in creating a Box2D program is to choose a scale factor from pixels to meters. All of the examples in the tutorial use a scale of 30. Normally, all further dimensions would need to be expressed in fractions of meters. The B2 library only uses pixels as a unit and converts pixels to meters, and vice versa, automatically. In B2, the width of boxes and circles is always the diameter.

Box2D worlds are populated with bodies. Bodies can be composed from many shapes, referred to as fixtures in Box2D. The shapes in a body can accessed by index, 0, 1, etc. A body can be **static**, which means that it is frozen in place and cannot be moved by forces. The more common choice is to create **dynamic** bodies, which are affected by gravity and other forces. B2 will automatically delete bodies that fly off the left/right side of the screen and bodies that fall through the floor. Bodies that rise above the top of the screen are allowed to fall back into view.

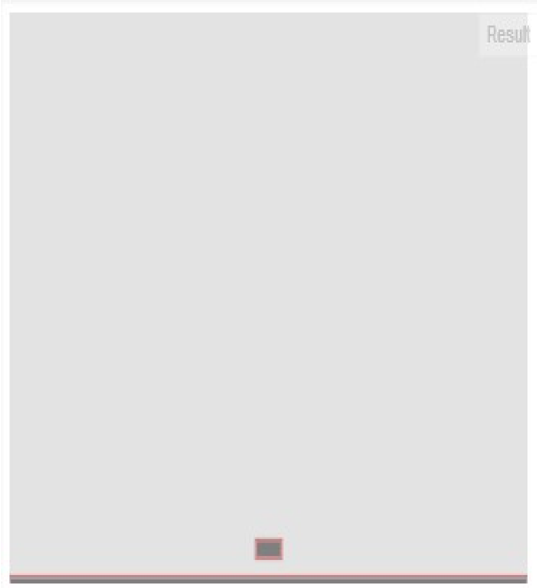
Boxes and Circles

The first example creates a static floor and then drops a dynamic box from the top of the screen. Notice that the box and floor have a red outline, which is drawn because "debug drawing" was enabled by passing "true" to b2Draw. The b2Draw routine only draws the shapes that are associated with a body. For debugging, it can be useful to view joint connections, which are drawn when the debug argument is true.

Also notice that the default colors are a not-very-exciting, gray. B2 extends the properties that are defined in Box2D with additional, hopefully useful, properties, such as "display", which supports user-draw code.

Static Floor, Dynamic Dropped Box
http://jsfiddle.net/bobcook/fawk0u7b/

```
function setup() {
   createCanvas(400, 300);
   b2newWorld(30, createVector(0, 9.8));  //scale and gravity
   // body type, dynamic/static, xy in pixels, wh in pixels
   new b2Body('box', false, createVector(width / 2, height - 2), createVector(width, 5)); // floor
   new b2Body('box', true, createVector(width / 2, 0), createVector(20,10));  // dropped box
}

function draw() {
   background(227);
   b2Update();  // MUST be present to update forces/position in Box2D
   b2Draw(true);  // default draw code for bodies, true/false controls debug drawing
}
```



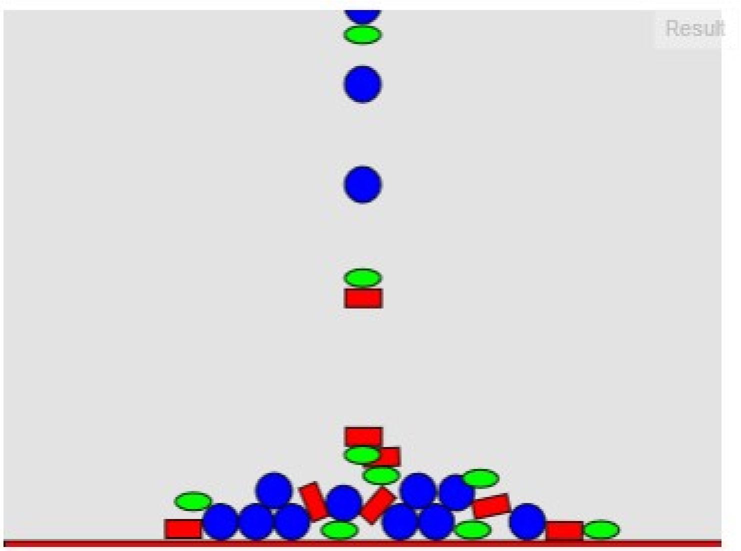
The second example illustrates user-draw code and introduces the circle shape. Circles work well for balls, rocks, and other objects that can be treated as perfectly round when calculating collisions. If the visible sprite is round but irregular (say, a boulder or an egg), you may wish to set the radius slightly smaller than the physical image size.

To add a little excitement, clicking the mouse will randomly create a new box (red), circle (blue) or ellipse (green). The user has the choice between totally drawing a shape or simply setting the properties and then calling b2Display, which uses the matching P5 function to complete the figure. Observe that even though we are drawing an ellipse, the easiest approximation to its shape is a box. The most accurate approximation would be a polygon.

Some properties, such as "xy" or "static", are defined for bodies. Since a body can be composed of many shapes, the properties, such as "display", "type" or "wh", of individual shapes must be accessed by calling a function with the shape-index as an argument. A body that is composed of only one shape has an index of 0.

Observe that shapes on any body can share draw routines or each shape can be "display"ed with different functions.

User-Draw Box/Circle/Ellipse Bodies
http://jsfiddle.net/bobcook/q247oj9m/



```
function setup() {
   createCanvas(400, 300);
   b2newWorld(30, createVector(0, 9.8));
   var body = new b2Body('box', false, createVector(width / 2, height - 2), createVector(width, 5));
   body.display(attr, 0);
}

function draw() {
   background(227);
   b2Update();
   b2Draw(false);  //no debugging
}

function mousePressed() {
   var body;
   if (random() < 0.3) {
      body = new b2Body('box', true, createVector(width / 2, 0), createVector(20,10));
      body.display(attr, 0);    //shape index 0
   } else if (random() < 0.6) {  //w is the circle's diameter
      body=new b2Body('circle', true, createVector(width / 2, 0), createVector(20,10));
      body.display(attr, 0);
   } else {
      body=new b2Body('box', true, createVector(width / 2, 0), createVector(20,10));
      body.display(attr1, 0);
   }
}

function attr(body, fixture, position) { --only set attributes
   if (body.type(0) == 'box') fill(255,0,0);
   else fill(0,0,255);
   b2Display(body, fixture, position);
}
```

```
function attr1(body, fixture, position) { --take total control
   fill(0,255,0);
   ellipse(position.x, position.y, body.wh(0).x, body.wh(0).y);
}
```

Edges and Polygons

The third example introduces edge and convex polygon shapes. Observe that the first two examples coded the floor as a static box. That choice gets unwieldy for bumpy surfaces. Remember that shapes are positioned at the center so trying to align the top corners to get an unbroken surface is a trial.

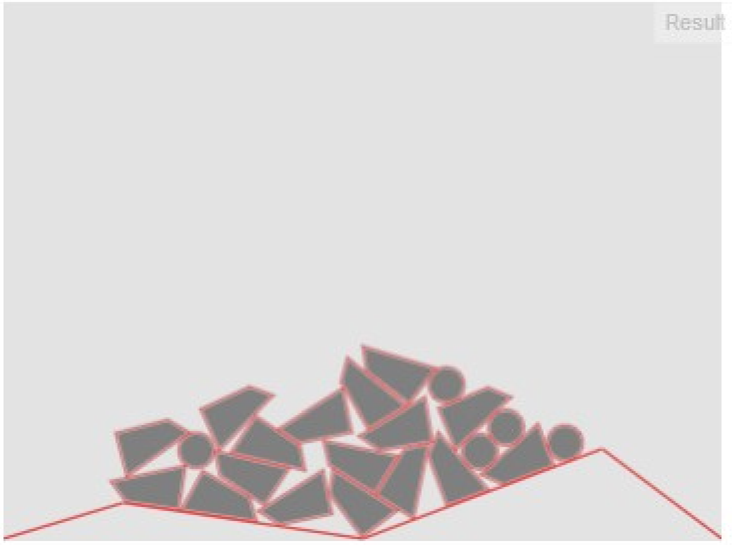
The static edge-shape solves the surface issue by connecting a list of lines to specify the boundaries of a world. The example defines a bumpy floor. Edge shapes are not restricted to 8 vertices and can contain concave dips. The vertices for both polygon and edge shapes are relative to the xy position of the body.

Click the mouse to drop circle or polygon shapes. The restitution of the circles is set high to illustrate different bounce values. Polygons are also specified as a list of lines; however, the last point is automatically connected to the first point. Polygon shapes can be created as static or dynamic bodies. Note that the **polygon coordinates must also be defined in counter-clockwise order**, and the resulting shape must be convex-only.

The optional arguments to the "b2Body" method are the first shape's density, friction, bounciness, and angle, which is the orientation in radians relative to its center.

- **density** is multiplied by the area of a body’s shape to determine its mass. This parameter is based on a standard value of 1.0 for water, so materials that are lighter than water (such as wood) have a density below 1.0, and heavier materials (such as stone) have a density greater than 1.0. The default value is 1.0.
- **friction** may be any non-negative value; a value of 0 means no friction and 1.0 means fairly strong friction. The default value is 0.5.
- **bounce** is the Box2d property known internally as “restitution”. It determines how much of an object’s velocity is conserved after a collision. Values greater than 0.3 are fairly “bouncy”. An object with a bounce value of 1.0 will rebound forever (i.e., if dropped to the ground, it will bounce back up to the height from which it was dropped). Bounce values higher than 1.0 are valid, but will cause strange behavior: objects will literally gain velocity with each collision until they fly off into space. The default value is 0.2, which is slightly bouncy.
- **angle** is the rotation of the box shape relative to its center. The default value is 0.0. Box2D only supports the rotation of boxes and polygons or edges.

Edge and Polygon Example
http://jsfiddle.net/bobcook/knukkxqx/



```
function setup() {
   createCanvas(400, 300);
   b2newWorld(30, createVector(0, 9.8));
   new b2Body('edge', false, v(width / 2, height - 2),
    [v(-width/2, 0), v(-width/3,-20), v(0,0), v(width/3,-50), v(width/2,0)]);
}

function draw() {
   background(227);
   b2Update();
   b2Draw(true);
}

function mousePressed() {
   var body;
   if (random() < 0.5) {                                                   //density, friction, bounce
      body=new b2Body('circle', true, v(width / 2, 0), v(20,10), 1, 0.5, 0.6);
   } else {
      body=new b2Body('polygon', true, v(width / 2, 0), [v(15,-15), v(-25,0), v(-15,10), v(15,10)]);
   }
}

function v(x, y) {
  return createVector(x, y);
}
```

b2Body Methods and Properties	
Methods	**Descriptions**
b2Body(type, isDynamic, xy, wh)	type 'box' 'circle' 'polygon' 'edge' xy is a P5 vector in pixels wh is a P5 vector or an array of vectors (edge,polygon)

b2Body(type, isDynamic, xy, wh,density, friction, restitution, angle)	angle is the rotation in radians and only affects the box shape.
b2Display(body, fixture, position)	helper function for user-draw code. Just set the attributes, such as fill color, then call to draw. Supports the image property scaled to wh, except for polygons. The user can store additional properties in bodies and shapes.
b2GetBodyAt(x, y)	returns null or the dynamic body that is located at pixel-position x/y. Used for mouse or finger-tap selection.
body.**toString**()	returns x/y velocity.x/y, both in pixels.
body.**destroy**()	removes the body and all attachments from the world. Bodies are automatically destroyed if their "life property is zero or if their position moves off the left, right, or bottom of the screen.
body.**addTo**(type, xy, wh, angle)	add additional shapes to a body. The angle argument is optional. xy is the offset from the parent's center.
body.**type**(index)	index is optional in all these functions. If omitted, zero is assumed as the shape number. Returns one of 'box' 'circle' 'polygon' 'edge'
body.**destroyShape**(index)	removes the shape from the body/world, has no effect on other indices.
body.**image**(img, index)	sets the image property of a shape to a P5 image.
body.**wh**(index)	returns the wh value set in b2Body.
body.**display**(func, index)	sets the user-draw function for a shape. Set to null to restore the default.
Properties	**Descriptions**
body.**static**	returns true for a static body, false for dynamic.
body.**active**	returns true if the body is alive, false if deletion is pending.
body.**visible**	get or set to true/false to control drawing.
body.**life**	get or set to number of time steps until destruction, default 10000000.
body.**density**	get or set density on all attached shapes.
body.**friction**	get or set friction on all attached shapes.
body.**bounce**	get or set restitution on all attached shapes.
body.**xy**	returns a P5 vector for the current position in pixels.
body.**angle**	returns the rotation of a body in radians.
body.**mass**	returns the weight in kilograms.
body.**centerOfMass**	returns a P5 vector for the mass center of a body in pixels.
body.**classOf**	returns 'b2Body'.

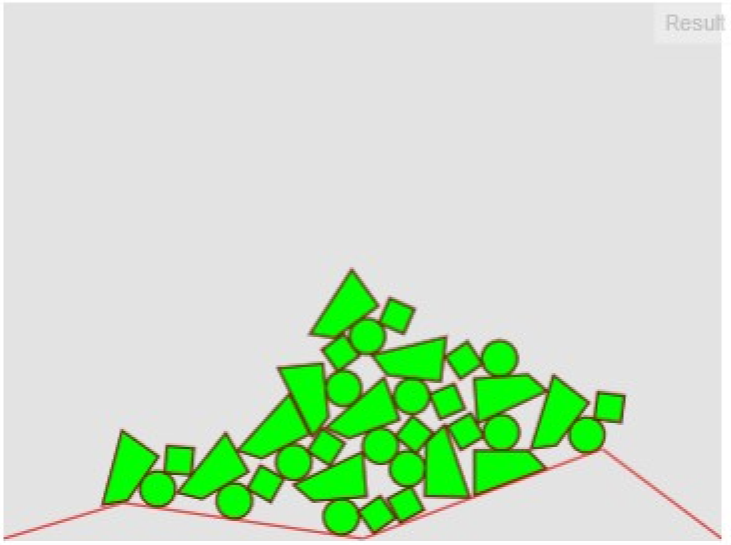
Multi-Shape Bodies

Using rectangles and circles as game objects is not very interesting. Players expect exciting images. The physics coders task is to partition images into convex polygons of no more than 8 vertices each. There are a number of applications, such as the Physics Editor (www.codeandweb.com/physicseditor), to automate the conversion. The shapes are used in the physics model, but the program displays the images.

In the next example, we define a body composed of a circle, a box, and a polygon. Remember that add-ons use coordinates to the host body's xy position. Further, each add-on also has an xy origin. This feature is useful for polygons as it allows shape definitions to be repositioned on the same, or different figures.

The "addTo" function adds shapes to a body. The arguments are an parent-offset xy-position, the width and height or an array of coordinates, and an optional angle, which only applies to boxes (as illustrated). The "display" property is set to a user-draw function to override the default fill color. Box2D does not require that a body's shapes touch each other. The shapes can have empty space between them, but the components always retain their defined orientation with a body's position. Observe that the box shape has been rotated relative to the body.

Multi-Shape Example
http://jsfiddle.net/bobcook/4rd29f09/



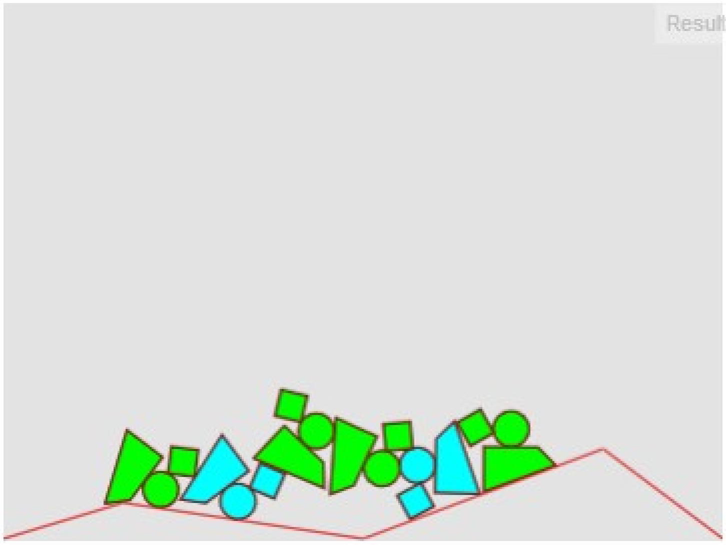
```
function setup() {
   createCanvas(400, 300);
   b2newWorld(30, createVector(0, 9.8));
   new b2Body('edge', false, v(width / 2, height - 2),
    [v(-width/2, 0), v(-width/3,-20), v(0,0), v(width/3,-50), v(width/2,0)]);
}

function draw() {
   background(227);
   b2Update();
   b2Draw(true);
}
```

```
function mousePressed() {
   var body = new b2Body('circle', true, v(width / 2, 0), v(20,10));
   body.display(foo);
   body.addTo('polygon', v(0, -20), [v(15,-15), v(-25,0), v(-15,10), v(15,10)]);
   body.display(foo, 1);
   body.addTo('box', v(20, 0), v(15, 15), PI/3);
   body.display(foo, 2);
}

function foo(body, fixture, position) {
   fill(0, 255, 0);
   b2Display(body, fixture, position);
}

function v(x, y) {
  return createVector(x, y);
}
```

Body Selection

The final body example illustrates how to convert mouse coordinates to a body reference. The first time that a body is touched, its color will change to blue. The second touch on the same body will delete it. There are many game applications for mouse selection. Observe that the code adds the user-defined "touched" property to each body.

Body Selection Example
http://jsfiddle.net/bobcook/te2eo0q0/



```
//only the changed code
function mousePressed() {
   var b = b2GetBodyAt(mouseX, mouseY);
   if (b != null) {
      if (b.touched++ > 0) b.destroy();
      return;
```

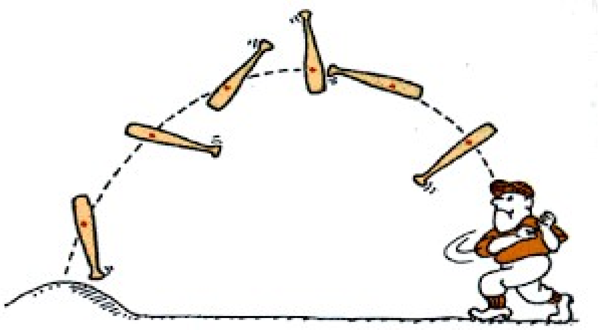
```
  }
  // same as before
}

function foo(body, fixture, position) {
  fill(0, 255, body.touched!=0 ? 255 : 0);
  b2Display(body, fixture, position);
}
```

Applying Force

In the previous examples, gravity was the primary force, although body collisions also imparted forces. In order to model the physics of projectiles, the shooting-throwing-slinging mechanism must be able to apply a force at one or more points on a body. Similarly, driving a car requires that forces be applied to the chassis.

The default point in Box2D at which force is applied is the **center of mass**. For rigid bodies, the center of mass is the centroid. Remember that in Box2D the mass of a body equals the density times its area. Informally, the **centroid** is the point at which a cardboard cut-out of a body could be perfectly balanced on the tip of a pencil. The centroid is important because a force applied at that point moves an object in the direction of the force. Pushing a toy block at an edge will rotate it. Pushing a toy block in the center of a side will slide it backwards.

Center of Mass



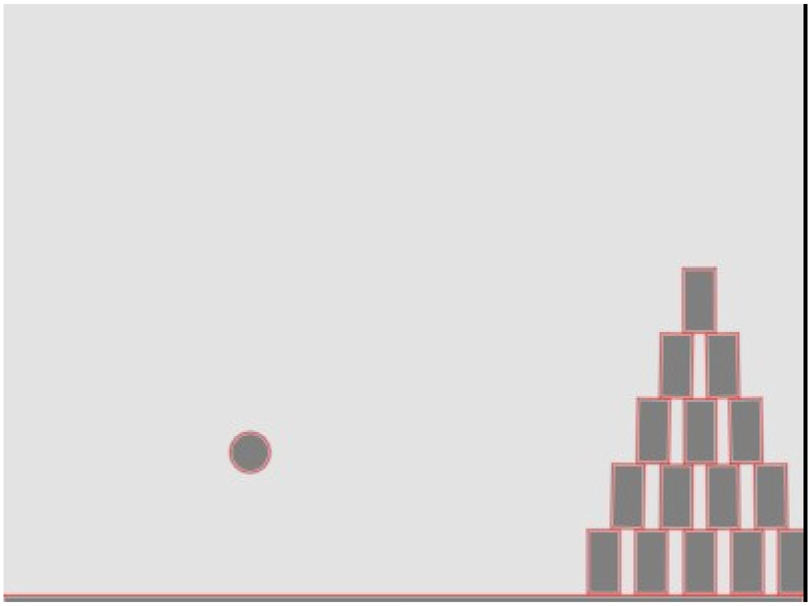
The Box2d body API defines a number of properties and methods to apply and regulate forces. The first example only uses one of the methods "applyForce". The object:applyForce() method adds (vector-wise) the specified x and y components of a force at a given point in or on an object. If the target point is the body's center of mass, it will tend to push the body in a straight line. If the target point is offset from the center, the body will spin around its center of mass. For symmetrical objects, the center of mass and the center of the object will have the same position. The force is not applied immediately but rather is factored into the physics engine's next time step and combined with other forces, such as gravity. Just as a car glides to a halt if its motor is turned off, a force that is just applied once will not keep a body moving for long.

Think of the "xyForce" vector as the steering wheel of a car that is omni-directional and the "power" value as the gas pedal.

body.**applyForce**(xyForce, power)

In the cannon example, we stack cans then the player shoots cannon balls to try to knock them all off the wall. The following code draws the scene and stacks the cans. When the mouse is pressed, a circular cannon ball is created and fired in the direction of the mouse click. The ball is set to a higher density than the cans, which would be the case in real life.

Cannon Ball Shooter
http://jsfiddle.net/bobcook/8djgkc24/

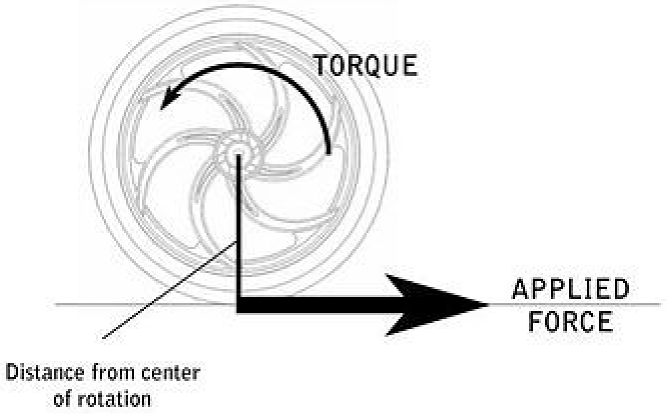


```
function setup() {
   createCanvas(400, 300);
   b2newWorld(30, createVector(0, 9.8));
   base(width / 2, height - 2, width, 5);
           N    X            Y        W   H
   pyramid(5, width*3/4, height - 10, 16, 32);
}

function draw() {
   background(227);
   b2Update();
   b2Draw(true);
}

function mousePressed() {
   var b = shape('circle', 5, height - 8, 20, 20);
   b.density = 20;
   b.applyForce(createVector(1, (height-mouseY)/height*2), 300);
}

function base(x, y, w, h) {
   return new b2Body('box', false, createVector(x, y), createVector(w, h));
}

function shape(type, x, y, w, h) {
   return new b2Body(type, true, createVector(x || width / 2, y || 0), createVector(w || 20, h || 20));
}

function pyramid(n, x, y, w , h) {
    // Build a pyramid with an n-can base
  var xy = createVector(x,y);
  var wh = createVector(w,h);
  while (n > 0) {
    var i = 0;
    var _xy = xy.copy();
    while (i < n) {
      var b = shape('box', _xy.x, _xy.y, wh.x, wh.y);
      b.density = 5;
      _xy.x += wh.x / 2 + wh.x;
      i++;
    }
    n--;
    xy.x += wh.x/2+wh.x/4;
    xy.y -= wh.y*3/2;
  }
}
```

Shooting an object is easy. Just touch the screen and a new ball is created at the y-coordinate height, and then thrown with a power of 300. The code could easily be turned into a game by gradually increasing the number and types of cans. Remember that off-screen balls and cans are deleted automatically.

The following table lists the properties and methods for a Box2D body. Forces are specified either in pixels for straight-line application or in radians for rotation. Remember that the P5 radians() function will convert degrees to radians.

Body Force Methods and Properties	
Methods	**Descriptions**
body.**applyForce**(xyForce, nPower)	Applies the vector force (multiplied by nPower) in pixels/second over multiple time steps at the center of mass taking into account other forces, such as gravity.
body**.applyImpulse**(xyForce, nPower)	Adds in the vector force (multiplied by nPower) in pixels/second immediately. If there are, for example, 30 time steps per second, this means the effect will be magnified by a factor of 30 over applyForce.
body.**applyTorque**(radians)	Applies a rotational force to the body over multiple time steps. Positive values will result in clockwise rotation; negative values will result in counter-clockwise torque. The body will rotate about its center of mass.

	
body.**applyAngularImpulse**(radians)	Adds a rotational force immediately.
body.**sensor**(bool, index)	Turns the sensor property of a shape on (true) or off (false).
body.**isSensor**(index)	Returns the sensor property of a shape.
Properties	**Descriptions**
body.**gravityScale**	Sets the effect of gravity on the body. Setting it to 0 makes the body float, even if other objects in the simulation are subject to normal gravity. The default value is 1.0 (normal gravity).
body.**bullet**	Get or set to true/false. Prevents fast-moving objects from passing through thin barriers.
body.**velocity**	Returns the xy-velocity in pixels per second.
body.**collision**	Gets or sets the collision listener function, func(body1,body2).
body.**categories**	A category defines which bodies collide. The default category is 1. A category of 0 does not collide. Other category selectors are 2,4,8,16,32,64,128,256,512,1024,2048,4096,8192,16384,32768. A body can be a member of multiple categories e.g. 256+8+1.
body.**collidesWith**	Lists the categories with which this body collides. Colliding with all categories is the default. Zero indicates no collisions. Specify as 2048+32+1 for readability.

The "bullet" property is set on any body that moves "fast" relative to the obstacles that it might encounter. Basically, if the velocity step in pixels is larger than the width of an obstacle, collisions might not be detected. The game programming term is **tunneling**. Try running the following sample code with, and without, the bullet property set. Observe that that it takes a much lower "impulse" to shoot the bullet. Try switching between Force and Impulse.

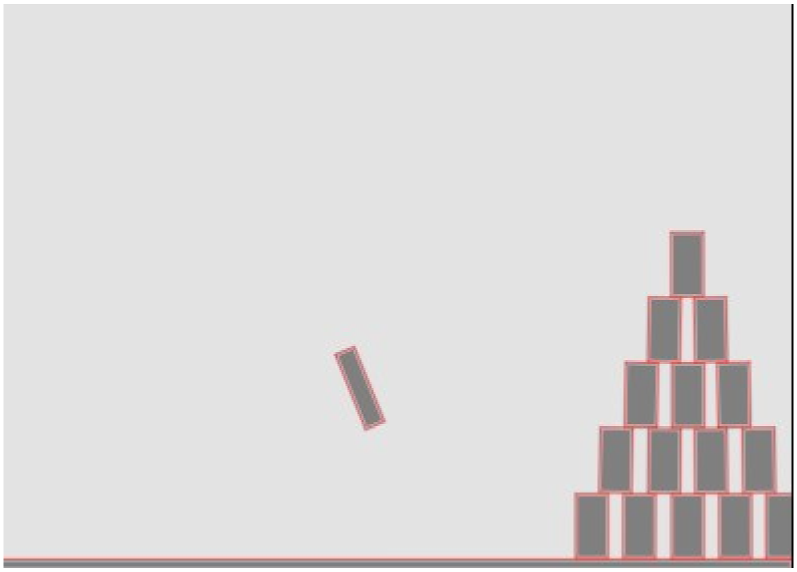
Tunneling Example

http://jsfiddle.net/bobcook/cy8rr9ep/

```
function setup() {
    createCanvas(400, 300);
    b2newWorld(30, createVector(0, 9.8));
    base(width / 2, height - 2, width, 5);
    // Stacked Boxes
    for (var i = 0; i < height / 2 / 25; ++i) {
        var b = shape('box', width * 2 / 3, height - 5 - i * 10 - 20, 5, 10);
    }
}

function draw() {
    background(227);
    b2Update();
    b2Draw(true);
}

function mousePressed() {
    var b = shape('circle', 20, height - 8, 3, 3);
    b.bullet = true; //collision not detected without the flag
    b.applyImpulse(createVector(1, 0), 0.4);
}

function base(x, y, w, h) {
    return new b2Body('box', false, createVector(x, y), createVector(w, h));
}

function shape(type, x, y, w, h) {
    return new b2Body(type, true, createVector(x || width / 2, y || 0), createVector(w || 20, h || 20));
}
```

Torque Example

http://jsfiddle.net/bobcook/8djgkc24/



```
function mousePressed() {
    var b = shape('box', 5, height-80, 40, 10);
    b.density = 20;
    b.gravityScale = 0;  //unaffected by gravity
    b.applyForce(createVector(1, (height-mouseY)/height*2), 300);
    b.applyTorque(550.0);  //positive is clockwise
}
```

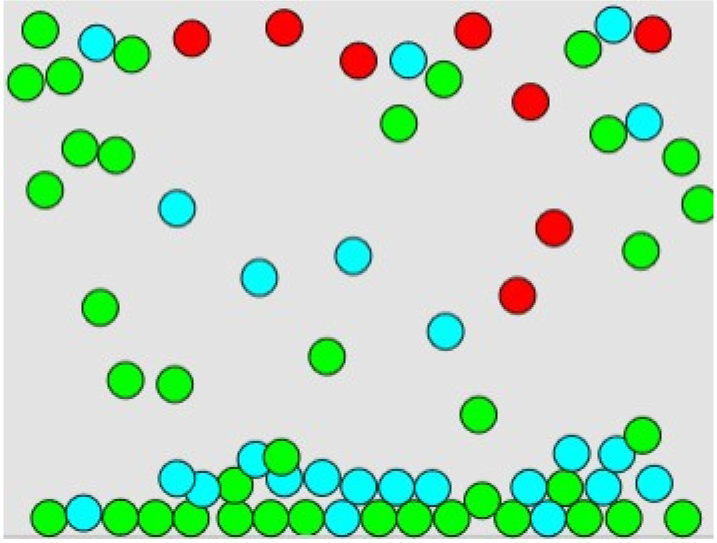
Collisions

A typical game implements bodies that collide, such as characters and swords, as well as bodies that should not collide, such as the particles in an explosion or a rider on a bicycle. By default, all objects, both static and dynamic, collide with each other.

The library defines a "collision" property on bodies that can be set to a collision-listener function. Each body can attach its own listener, or bodies can share a function.

In the next example, circles are dropped at random X locations. When a collision is detected between two bodies, the listener is invoked. The color on one body is set to green and the color of the other body is set to cyan. Running the example illustrates that the color of a colliding body can oscillate between cyan and green; in other words, there is no fixed order in which body is number one or two.

Collision Listener Function
http://jsfiddle.net/bobcook/ajewna8r/



```
function setup() {
    createCanvas(400, 300);
    b2newWorld(30, createVector(0, 9.8));
    new b2Body('edge', false, v(width / 2, height - 2), [v(-width/2, 0),
    v(width/2,0)]);
}

function draw() {
    background(227);
    var b = new b2Body('circle', true, v(random(12,width-12),12), v(20,20));
    b.color = color(255,0,0);
```

```
  b.display(drawer);
  b.collision = bump;
  b2Update();
  b2Draw(false);
}

function drawer(body, shape, position) {
  fill(body.color);
  ellipse(position.x, position.y, 20, 20);
}

function bump(body1, body2) {
  body1.color = color(0,255,0);
  body2.color = color(0,255,255);
}

function v(x, y) {
 return createVector(x, y);
}
```

It is a common practice in programming to add redundant features because they provide a performance advantage. Such is the case with collision categories.

The programmer could resolve all collision tests in the "bump" function as listed in the previous example. However, it is more efficient to let the Box2D library "filter" unwanted collisions before they are propagated to the user.

Box2D provides 16 collision categories (1, 2, ... , 32768). Any body can be a member of any, all, or none of the categories. It is recommended to list the categories as a sum, such as 8+2+1. The default category is one for both static and dynamic bodies.

The "collidesWith" property of a body specifies the list of categories with which it collides. Again, it is recommended to list the categories as a sum from high to low.

Box2D Collision Formula
IF body1.categories **matchesOneOf** body2.collidesWith **OR**
body2.categories **matchesOneOf** body1.collidesWith **THEN** CALL_USER_CODE

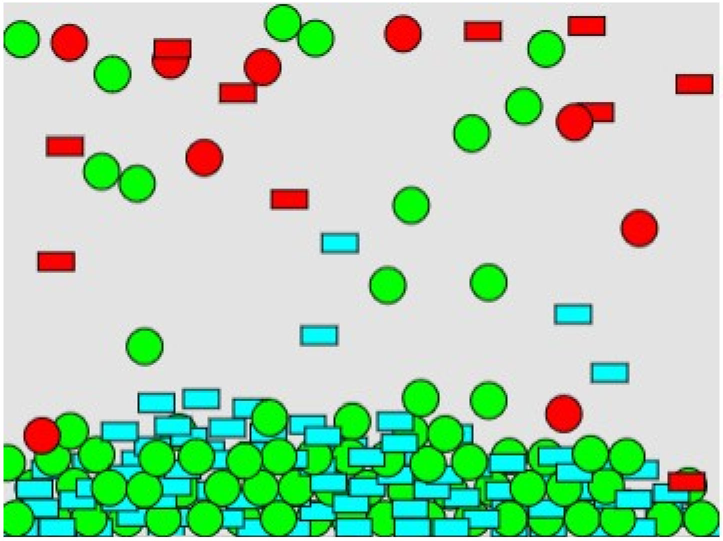
Category 4 and 8 Collision Options

Categories	CollidesWith	Result
body1 = 4	4	only collides with other 4s
body1 = 4	8	collides with 8s but not 4s
body1 = 4	8+4	collides with 8s and 4s

If two bodies touch but do not "collide", Box2D will allow the shapes to pass through each other or to overlap with no bounce or force calculation. Collision can also be asymmetric; that is, body1 can collide with the category of body2 without requiring that body2 also specify a collision.

The next example creates both boxes (category 2) and circles (category 4). Both shapes collide with themselves and the ground (remember default of 1), but not each other. Observe that both shapes form overlapping pyramids. As in the previous example, red shapes change color to green or cyan upon a collision.

Collision Categories
http://jsfiddle.net/bobcook/0tdnomtk/



```
function setup() {
   createCanvas(400, 300);
   b2newWorld(30, createVector(0, 9.8));
   new b2Body('edge', false, v(width / 2, height - 2), [v(-width/2, 0), v(width/2,0)]);
}

function draw() {
   background(227);
   var b;
   if (random() < 0.5) {
     b = new b2Body('circle', true, v(random(12,width-12),12), v(20,20));
     b.categories = 4;
     b.collidesWith = 4+1;
   } else {
     b = new b2Body('box', true, v(random(12,width-12),12), v(20,10));
     b.categories = 2;
     b.collidesWith = 2+1;
   }
   b.color = color(255,0,0);
   b.display(drawer);
   b.collision = bump;
   b2Update();
   b2Draw(false);
}
```

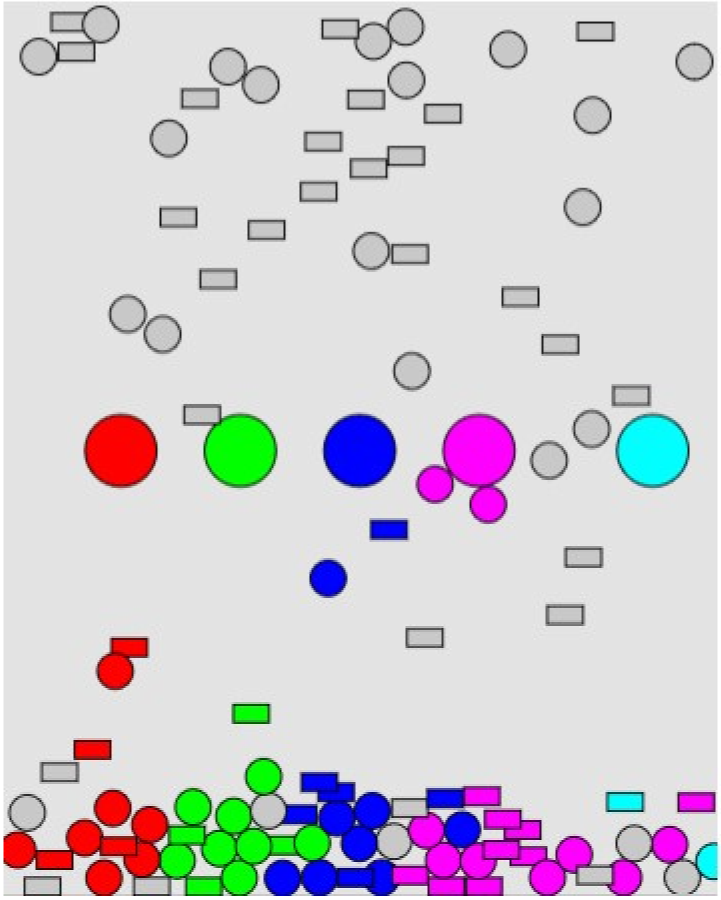
```
function drawer(body, shape, position) {
   fill(body.color);
   if (body.type()=='circle') ellipse(position.x, position.y, 20, 20);
   else rect(position.x, position.y, 20, 10);
}

function bump(body1, body2) {
   var col;
   if (body1.type() == 'circle') col = color(0,255,0);
   else col = color(0,255,255);
   body1.color = col;  body2.color = col;
}

function v(x, y) {
  return createVector(x, y);
}
```

Sensors

Sensors are a great feature of Box2D. A sensor calls a collision listener when a collision occurs; however, the physical effects of the collision are not applied. For example, consider a car's headlights. They illuminate obstacles in front of a car, but the light does not trigger a physical collision. In addition to headlights, sensors can be used for airplane radar, bug feelers, game proximity triggers, and many other applications.

In the sensor example, boxes and circles are dropped from the top of the screen. As the shapes pass through the colored sensor circles, the collision listener assigns a new color.

Sensor Example
http://jsfiddle.net/bobcook/ov5n9pqt/



```
function setup() {
   createCanvas(400, 500);
   b2newWorld(30, createVector(0, 9.8));
   new b2Body('edge', false, v(width / 2, height - 2), [v(-width/2, 0), v(width/2,0)]);
   create(6, color(255,0,0));
   create(3, color(0,255,0));
   create(2, color(0,0,255));
   create(1.5, color(255,0,255));
   create(1.1, color(0,255,255));
}

function create(fraction, col) {
   var b = new b2Body('circle', false, v(width / fraction, height/2), v(40, 40));
   b.display(drawer);
   b.color = col;
   b.sensor(true);
   b.categories = 2;
   b.collidesWith = 2;
   b.collision = bump;
}

function draw() {
   background(227);
   var b;
   if (random() < 0.5) {
     b = new b2Body('circle', true, v(random(12,width-12),12), v(20,20));
   } else {
     b = new b2Body('box', true, v(random(12,width-12),12), v(20,10));
   }
   b.categories = 2;
   b.collidesWith = 2+1;
   b.color = color(200, 200, 200);
   b.display(drawer);
   b2Update();
   b2Draw(false);
}

function drawer(body, shape, position) {
   fill(body.color);
   if (body.type()=='circle') ellipse(position.x, position.y, body.wh().x, body.wh().x);
   else rect(position.x, position.y, 20, 10);
}

function bump(body1, body2) {
   var col;
   if (body1.isSensor()) body2.color = body1.color;  //transfer color
   else body1.color = body2.color;
}

function v(x, y) {
  return createVector(x, y);
}
```

Joints

The human body has joints everywhere. Bones are rigid bodies that are acted on by internal forces applied by muscles and by external forces, such as gravity, and those applied by the environment. There are many different kinds of human joints.

Box2D implements a number of different joints. We only discuss some of the options. The examples are constructed with the goal of providing cut-and-paste code that the reader can use to model common game physics. All Box2D joints are created with the "b2Joint" method.

Joint Creation and Methods		
b2Joint(typeName, bodyA, bodyB, properties) typeName = distance, mouse, pulley, revolute, rope, wheel	**Properties** {name: value, ...}	adds a joint to bodyA. A body can have multiple joints, which are indexed 0..n. Returns the index number of the joint.

distance, rope	**separation**: inPixels	rope length for distance and rope joints.
distance, mouse, wheel	**frequency**: inHz	springiness (0 to 5) of distance, mouse and wheel joints.
distance, mouse, wheel	**damping**: ratio	damping ratio (0 to 1) of spring for distance, mouse and wheel joints.
distance, mouse, pulley, revolute, rope, wheel	**xy**: pixelVector	absolute xy position for pivot or attachment point or the target position for a mouse joint.
mouse, wheel	**maxForce**: value	maximum force exerted by mouse joint to drag body to target.
revolute, wheel	**speed**: radians/Sec.	wheel and revolute joint motor speed.
revolute, wheel	**enable**: true/false	turn on/off wheel and revolute joint motor.
revolute, wheel	**maxTorque**: value	controls how fast a motor comes up to speed or slows to a stop.
pulley	**xyb**: pixelVector	pulley joint, pivot point for bodyB.
pulley	**lengthA**: pixels	distance from pulley for bodyA.
pulley	**lengthB**: pixels	distance from pulley for bodyB.
pulley	**ratio**: value	scales the pulley force by a ration such that lengthA + ratio * lengthB <= constant.
body.**destroyJoint**(index)		as with other body methods, if index is omitted, zero is assumed.
body.**motorOn**(true/false, index)		motors can only be attached to revolute and wheel joints.
body.**isMotorOn**(index)		
body.**motorSpeed**(newValue, index)		
body.**getMotorSpeed**(index)		
body.**maxMotorTorque**(newValue , index)		
body.**getMaxMotorTorque**(index)		
body.**setTarget**(pixelVector, index)		body attached to mouse is dragged toward the target location.

Mouse Joint

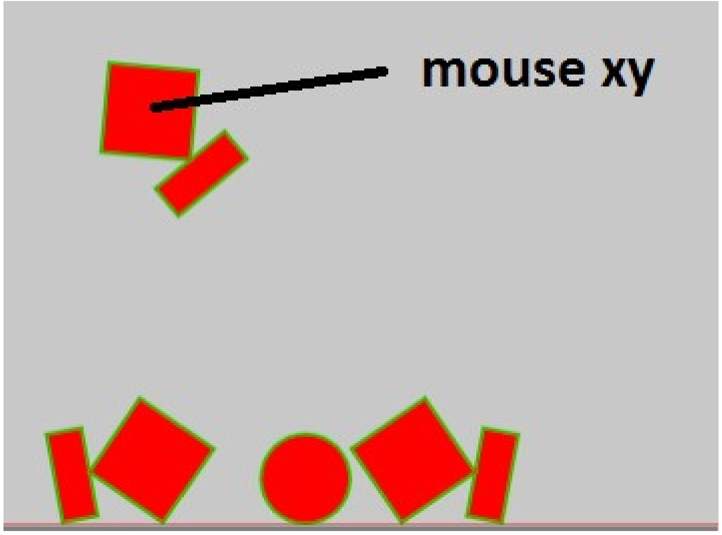
A mouse joint creates an elastic connection between an object and your mouse/finger, which makes the object attempt to follow the drag-path. But since the object remains under simulation, its motion may be stopped by other bodies in the world. Also, an object will rotate realistically under gravity when "picked up by one end".

The action of this joint is not automatic. The location of its attached body must be updated by calling setTarget(xy). The method can be called from a mouse/touch routine.

The overall speed or "lag" of the joint depends on the force exerted, which is stored in the maxForce property. By default, the attribute is set to 1000 times the mass of the body, which allows for fairly rapid dragging.

In the example, the selected shape follows the mouse, or touch, motion until the contact is released. When the mouse button is released, the shape drops to the ground.

Mouse Joint Example
http://jsfiddle.net/bobcook/fpcy09a0/



```
var mouse = null;

function setup() {
   createCanvas(400, 300);
   b2newWorld(30, createVector(0, 9.8));
   new b2Body('box', false, v(width / 2, height - 2), v(width, 5));
}

function draw() {
   background(200, 200, 200);
   b2Update();
   if (mouse != null) mouse.setTarget(v(mouseX, mouseY));
   b2Draw(true);
}

function keyTyped() {
   var t;
   if (key == 'b') {
       t = new b2Body('box', true, v(width / 2 - 2, 0), v(50, 20));
       t.addTo('box', v(0, -70 / 2 - 20 / 2), v(50, 50), PI / 4);
       t.display(foo, 1);
       t.applyAngularImpulse(10);
   } else if (key == 'c') {
       t = new b2Body('circle', true, v(width / 2, 0), v(50, 50));
   }
   t.display(foo);
}
var foo = function (body, fixture, pos) {
   fill(255, 0, 0);
   stroke(0, 255, 0);
   strokeWeight(2);
   b2Display(body, fixture, pos);
}

   function mousePressed() {
       var b = b2GetBodyAt(mouseX, mouseY);
       if (b == null) return;
       mouse = b;
       b2Joint("mouse", null, b, {
           xy: v(mouseX, mouseY)
       });
   }

   function mouseReleased() {
       if (mouse != null) mouse.destroyJoint();
       mouse = null;
   }

   function v(x, y) {
       return createVector(x, y);
   }
```

Revolute Joint

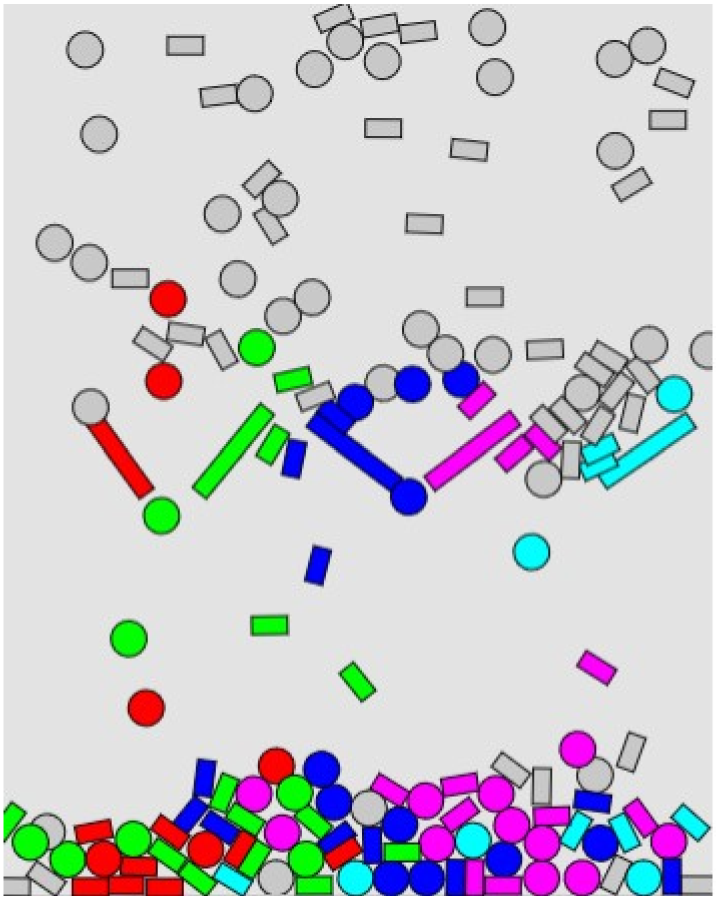
A revolute, or pivot, joint can model a hinge, a pin, or an axle that connects one body to another. The point of connection must be specified. It would not make sense to connect two static objects with a pivot joint, but any other combination has uses. The body can swing freely around the pivot point.

The really exciting feature of revolute joints is that they can have built-in motors. As a result, the motors can power body parts, flippers or any other mechanized device. Revolute joints can also be linked to create flexible structures such as chains or bridges.

By default, the motors have a fairly weak maximum torque, thus setting motorSpeed may appear to have little visible effect. Therefore, you should generally set maxMotorTorque to a high value (such as 10000) if you are trying to move any significant mass.

The following example updates the colored sensor example to create five colored, rotating flippers. When the falling shapes touch a flipper, the color changes. Shapes can be flipped into the air from one sensor to another before falling to the ground. Try un-commenting the sensor property, which will cause shapes to pass through but still change color if touched. Try changing the motor speed and torque values. Observe that if a motor is disabled, the flipper will still spin, but only in reaction to collision forces.

Revolute Joint Example
http://jsfiddle.net/bobcook/750q8oz8/



```
//ONLY CODE THAT CHANGED FROM SENSOR EXAMPLE
function create(fraction, col) {
   var b = new b2Body('box', true, v(width / fraction, height/2), v(60, 10));
   b.display(drawer);
   b.color = col;
   //b.sensor(true);
   b.categories = 2;
   b.collidesWith = 2;
   b.collision = bump;
   b.bumper = true;
   var c = new b2Body('circle',false,v(width/fraction,height/2),v(10,10));
   c.categories = 0;
   c.collidesWith = 0;
   b2Joint("revolute",c,b,{speed: PI*2, maxTorque:10000, enable:true});
}
```

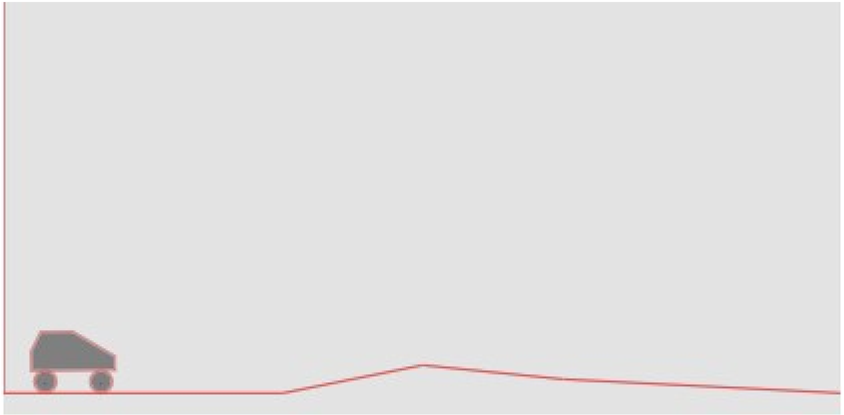
```
function drawer(body, shape, position) {
   fill(body.color);
   b2Display(body, shape, position);
}

function bump(body1, body2) {
   if (body1.bumper) body2.color = body1.color;
   else body1.color = body2.color;
}
```

Wheel Joint

A wheel joint can model a wheel that is connected to a body by a shock absorber. The point of connection must be specified as an offset that is relative to the center of the vehicle.. As with a revolute joint, a motor can be enabled to drive a vehicle forwards, or backwards. Since Box2D simulates a 2D world, typically only two wheels are attached.

The following car example attaches two wheel joints (with motors) to a vehicle. Typing 'd' or 'a' will drive the vehicle forwards or backwards, respectively.

Wheel Joint Example
http://jsfiddle.net/bobcook/k9d7tohn/



```
var hz = 4;
var zeta = 0.7;
var speed = 0;
var car;

function setup() {
   createCanvas(600, 300);
   b2newWorld(30, v(0, 9.8));
   var ground = new b2Body('edge', false, v(0, height - 20), [v(0, -height), v(0, 0), v(200, 0), v(300, -20),
v(400, -10), v(600, 0), v(600, -height)]);
   car = new b2Body('polygon', true, v(50, height - 55), [v(-30, 10), v(30, 10), v(30, 0), v(0, -18), v(-23, -18),
v(-30, -4)]);
   var wheel1 = shape('circle', 30, height - 35, 16); //back
   wheel1.friction = 0.9;
   var wheel2 = shape('circle', 70, height - 35, 16); //front
```

```
    wheel2.friction = 0.9;
    b2Joint('wheel', car, wheel2, {
        xy: v(70, height - 35),
        axis: v(0, 1),
        frequency: hz,
        damping: zeta,
        maxTorque: 20,
        speed: 0,
        enable: true
    });
    b2Joint('wheel', car, wheel1, {
        xy: v(30, height - 35),
        axis: v(0, 1),
        frequency: hz,
        damping: zeta,
        maxTorque: 20,
        speed: 0,
        enable: true
    });
}

function draw() {
    background(227);
    b2Update();
    b2Draw(true);
}

function keyTyped() {
    var speed = 0;
    if (key == 'd' || key == 'D') speed = PI * 12;
    if (key == 'a' || key == 'A') speed = -PI * 12;
    car.motorSpeed(speed);
    car.motorSpeed(speed, 1);
}

function shape(type, x, y, w, h) {
    return new b2Body(type, true, v(x || width / 2, y || 0), v(w || 20, h || 20));
}

function v(x, y) {
    return createVector(x, y);
}
```

Rope, Distance, and Pulley Joints

The three joints discussed in this Section are the rope, distance, and pulley joints. Again, each joint connects two objects; however, the objects are not overlapping, but are separated at a distance.

The Rope joint connects two objects with an invisible rope that connects the two center points. The distance between the centers is the length of the rope. The connection is not elastic so it can be shrunk, but is very resistant to stretching.

```
var rope1 = b2Joint("rope", point, circle, {separation: 140} )
```

The Distance joint is basically a Rope joint with lots of tuning parameters. The default length of the invisible rope is the distance between the two anchor points. However, a length property is defined for Distance joints so that the length can be set shorter or longer than the original distance. If one of the body types is static, the other body will move from its initial location to one that has the right distance from the first body's anchor point.

```
var dist = b2Joint("distance", point, circle, {separation: 80, frequency: 0.7, damping: 0.1} );
```

Another cool feature of Distance joints is that they can be springy; that is, the invisible rope can shorten and lengthen at a certain frequency like a weight on a rubber band. The two control properties are "frequency" and "damping".

> "The distance joint can also be made soft, like a spring-damper connection. Softness is achieved by tuning two constants in the definition: frequency and damping ratio. Think of the frequency as the frequency of a harmonic oscillator (like a guitar string). The frequency is specified in Hertz. Typically the frequency should be less than half the frequency of the time step. So if you are using a 60Hz time step, the frequency of the distance joint should be less than 30Hz. The reason is related to the Nyquist frequency. The damping ratio is non-dimensional and is typically between 0 and 1, but can be larger. At 1, the damping is critical (all oscillations should vanish)." [from Box2D]

At a frequency of zero or a damping value of one, a distance joint acts like a rope. Otherwise, forces on the bodies will induce a bouncing effect. The first bounce may even "shoot" one of the objects past the other anchor point. The damping value governs how quickly the bouncing slows down. Higher values will slow objects faster.

Pulleys have been used for thousands of years to make it possible to lift heavy loads through force multiplication. By using the right pulley configuration, 50kg of weight can be lifted by applying 10kg of force. With Box2D, complicated block-and-tackle pulley structures are unnecessary; only the force ratio needs to be specified.

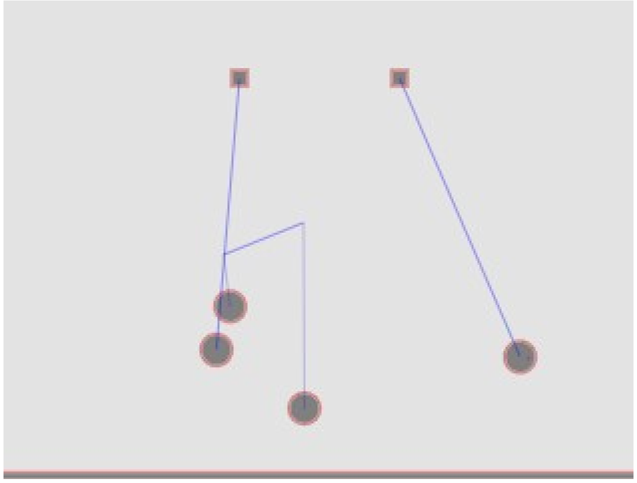
A Pulley joint attaches two bodies with an invisible rope that tries to keep its length constant: if one body is pulled down, the other one will move up. The b2Joint method is a bit complicated because it must specify two bodies and two stationary anchor points in world coordinates for each side of the rope to hang from.

```
joint = b2Joint( "pulley", bodyA, bodyB, {
      xy: createVector(140, 160),
      xyb: createVector(190, 140),
      lengthA: 30,
      lengthB: 40,
      ratio: 0.6
   })
```

The ratio argument simulates a block and tackle arrangement in which one side of the rope moves faster than the other. By default, this ratio is 1.0, which models a simple pulley. Each body can swing and rotate on its end of the invisible rope if forces are applied. Note that trying to pull one of the bodies past its anchor point results in bizarre behavior.

The example, which is illustrated in the next Figure, implements an instance of each of these joint types. Further, each of the circles can be touch-dragged to view the effect of different forces on the joints. The Figure is drawn in "debug" mode to show the movements of the invisible ropes.

Distance, Rope, Pulley Joint Example
http://jsfiddle.net/bobcook/81emgjnL/

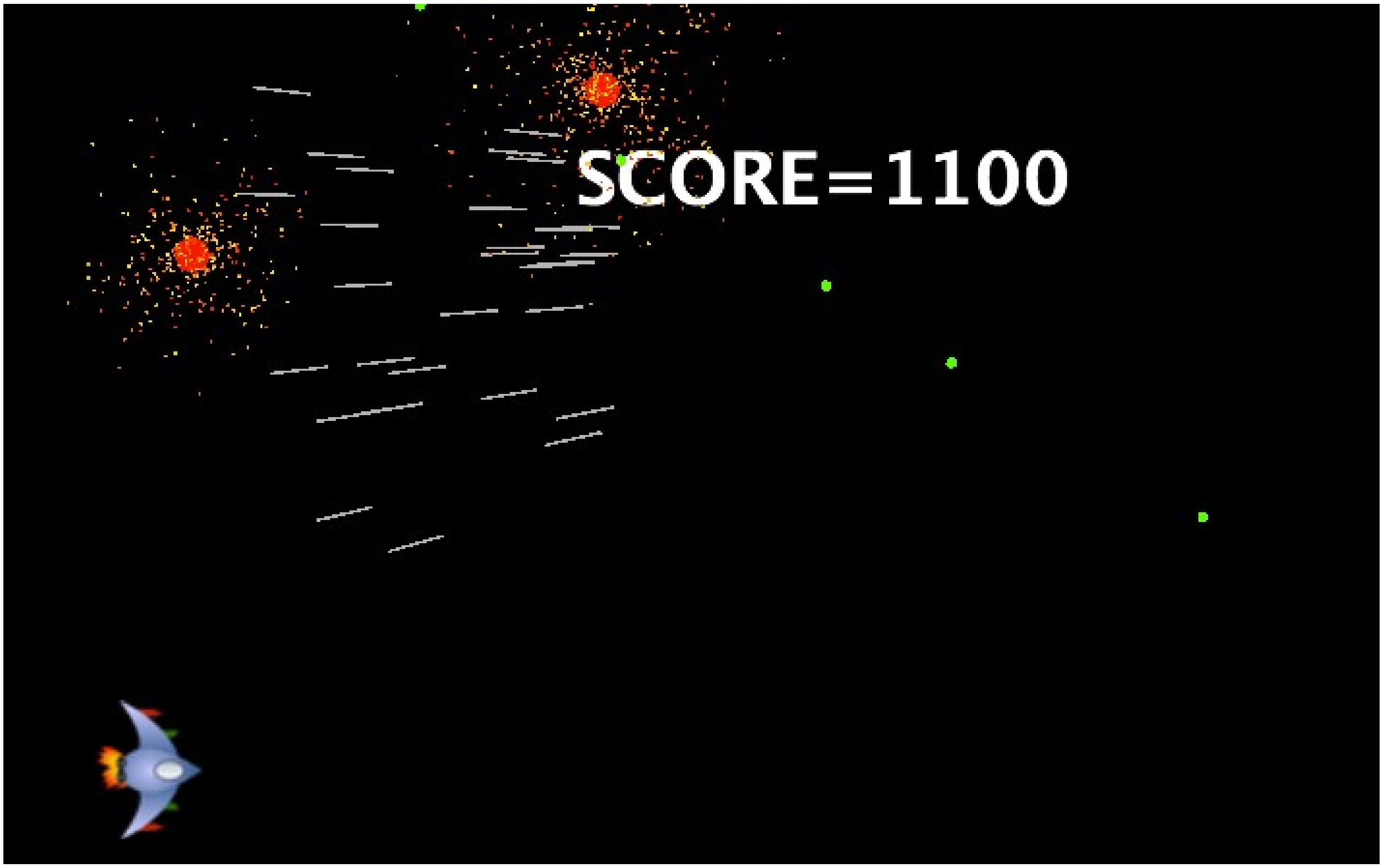


```
var mouse = null;
var mouseIndex;

function setup() {
   createCanvas(400, 300);
   b2newWorld(30, v(0, 9.8));
   new b2Body('box', false, v(width / 2, height - 2), v(width, 5));
   var b1 = base(250, 50);
   var b2 = circle(250, 80);
   b2Joint("distance", b1, b2, {
      separation: 80,
      frequency: 0.7,
      damping: 0.1
   });
   b1 = base(150, 50);
   b2 = circle(150, 100);
   b2Joint("rope", b1, b2, {
      separation: 140
   });
   b1 = circle(140, 180);
   b2 = circle(190, 180);
   b2Joint("pulley", b1, b2, {
      xy: v(140, 160),
      xyb: v(190, 140),
      lengthA: 30,
      lengthB: 40,
      ratio: 0.6
   });
}

function draw() {
   background(227);
   b2Update();
   if (mouse != null) mouse.setTarget(v(mouseX, mouseY), mouseIndex);
   b2Draw(true);
}

function base(x, y) {
   return new b2Body('box', false, v(x, y), v(10, 10));
}

function circle(x, y) {
   return new b2Body('circle', true, v(x, y), v(20, 20));
}

function mousePressed() {
   var b = b2GetBodyAt(mouseX, mouseY);
   if (b == null) return;
   mouseIndex = b2Joint("mouse", null, b, {xy: v(mouseX, mouseY)});
   mouse = b;
}

function mouseReleased() {
   if (mouse != null) mouse.destroyJoint(mouseIndex);
   mouse = null;
}

function v(x, y) {
   return createVector(x, y);
}
```

Alien Attack

The Alien Attack game flies alien spacecraft from left to right on the screen. If a craft makes it through the defensive perimeter on the right side of the screen, the player loses 100 points. The player has an array of weapons on the right side of the screen. The weapons shoot projectiles toward locations as they are clicked/tapped by the player; however, each weapon has a fixed fire rate and reload time. If the player destroys a craft or knocks it off the screen, the player gains 100 points.

Alien Attack Game



The example embodies the design principle that we love best (step-wise refinement). The program is fairly simple; however, it utilizes two sophisticated classes (Fountain and Projectile), which were built and tested in the order listed before the main code was ever written.

Fountains

The Fountain class was a refinement of the earlier Particle class. I started the implementation by passing the shape names to the class but that soon proved inflexible and unwieldy. It was an example of trying to do too many things in one piece of code. Every addition of a shape required the insertion of the corresponding drawing code in the class.

The better solution was to pass the name of the shape-drawing code as an argument. The name is then used as an index into a table of function values. Users can now add new shape names and attach drawing routines without touching the Fountain code.

Particles are defined as a table. One of the notable features of the design is that the view of a particle is totally separated from its actions. The Particle "life" property is defined as a fraction from 0 to 1. The fraction can then be used to select different display attributes, such as color, over a particle's lifetime. Particles have location and size properties as well as linear and angular velocities.

Particle Table Definition

```
function Particle() {
  this.xVel = 0;
  this.yVel = 0;
  this.partSize = 0;
  this.x = 0;
  this.y = 0;
  this.life = 0;
  this.rotation = 0;
}
```

Fountains can be defined in a JSON file or string. The JSON specifies an array of named fountain definitions. After the JSON is loaded, the converted object is passed as an argument to the Fountain class' constructor to create new fountains. The following table illustrates the JSON properties.

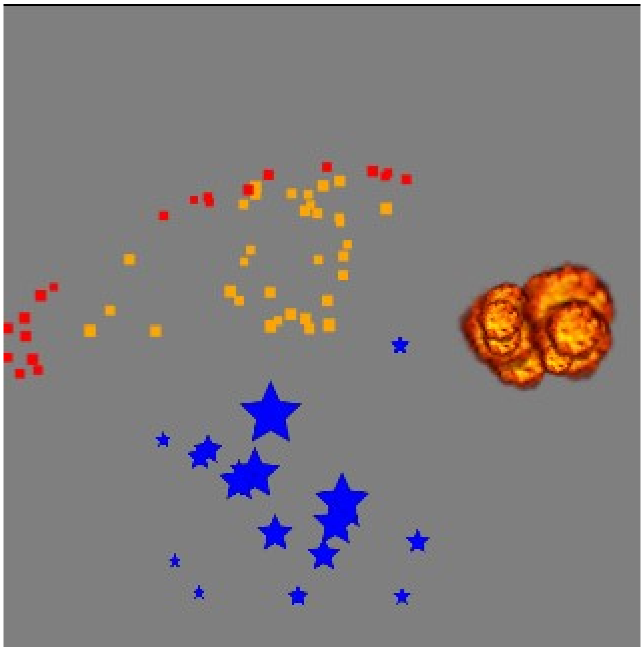
The Fountain definition is listed as a Fiddle that inputs a JSON Fountain string. Change the properties to become familiar with particle effects. To experiment with image-based particles, the example must be run on a PC or tablet. Particle generation is a sophisticated technical area. Our definition only includes the basics. See particle2dx.com for other options, which could be easily added.

JSON Fountain Properties

Name	Description	Example
name	unique name for this fountain	"name": "foo"
color	array of color names or [R,G,B,A] values. If more than one color is specified, the color changes in 1/length lifetime steps. Alpha values can be changed to fade particles in or out. Color should be white for images but other colors or alpha values can be listed as tints.	"color": ["orange", [255,0,0],"yellow","white","white","white"]
shape	string index into the fdisplay array to select a drawing function.	"shape": "rect"
gravity	added to yVelocity at each Step.	"gravity": 0.1
sizePercent	particle size is multiplied by this value at each Step. Fractions less than one shrink a particle, greater than one expands a particle.	"sizePercent": 0.999
angle	defines the spread of the particle stream in degrees.	"angle": [190,300]
speed	velocity is set by a random angle and multiplied by speed.	"speed": 5.5
limit	number of particles generated at one particle per Step.	"limit": 250
lifetime	number of Steps before the particle is deleted.	"lifetime": 200
size	defines the range of particle sizes.	"size": [4,12]
Optional		
rotation	angular velocity in degrees. If non-zero, the initial rotation is random(0, 360).	"rotation": 2

file	path to an image file. Images are scaled by partSize.	"file": "assets/fire.png"
x	sets particle x-origin to width/x.	"x": 2
y	sets particle y-origin to height/y.	"y": 2
edges	array of vectors that define counter-clockwise points (relative to xy) in a polygon	"edges": [[5,-5],[-5,-5],[0,5]]

JSON Examples



```
{
  "parts": [
   {
  "name": "foo",
  "color": ["orange","red","yellow","white","white","white"],
  "shape": "rect",
  "gravity": 0.1,
  "sizePercent": 0.999,
  "angle": [190,300],
  "speed": 4.0,
  "limit": 250,
  "lifetime": 200,
  "size": [4,6],
  "x": 2,
  "y": 2},
    {
  "name": "boo",
  "color": ["white"],
```

```
var defs;
var of, mf, sf;
function preload() {
  defs = loadJSON('assets/fountains2.txt');
}
function setup() {
  createCanvas(320, 320);
  of = new Fountain(defs, "foo");
  mf = new Fountain(defs, "boo");
  sf = new Fountain(defs, "soo");
}

function draw() {
  background(127);
  of.Draw();
  of.Create();
  of.Step();
  mf.Draw();
```

```
    "shape": "image",
    "file":  "assets/fireball.png",
    "gravity": 0.0,
    "sizePercent": 0.99,
    "angle": [0,360],
    "speed": 0.4,
    "rotation": 0.0,
    "limit": 20,
    "lifetime": 50,
    "size": [1,3],
    "x": 1.2,
    "y": 2},
    {
    "name": "soo",
    "color": ["blue"],
    "shape": "image",
    "file":  "assets/star.png",
    "gravity": 0.1,
    "sizePercent": 0.9,
    "angle": [0,360],
    "speed": 4.0,
    "rotation": 0,
    "limit": 50,
    "lifetime": 100,
    "size": [1,1],
    "x": 2,
    "y": 1.5}
    ]
}
```

```
  mf.Create();
  mf.Step();
  sf.Draw();
  sf.Create();
  sf.Step();
  noStroke();
  text(of.length, width - 60, 20);
  stroke(0);
}
```

Fountain Class Definition

Methods	
new Fountain(JSON table, name, x, y)	xy are optional and specify the pixel origin for the fountain. If omitted, JSON xy are used.
Create(x, y)	xy are optional and specify the pixel origin of this particle. Creates one particle unless the limit has been exceeded. Returns an instance of Particle.
Draw()	draws all particles in this fountain.
Step()	advances positions of all particles and deletes particles if their life is over.
Stop()	stops the fountain and deletes all particles.
Properties	
particles[]	array of active particles.

f	reference to the JSON table entry for this fountain.
colors[]	f.color array converted to P5 colors
x	pixel origin position
y	pixel origin position
n	number of particles to create.
draw	function to draw the particles, set from fdisplay[shape]
rotation	JSON value or 0, if not specified
image	reference to the loaded P5 image
length	number of active particles
left	number of particles still to create.
done	true if all particles have been created and deleted

Sample fdisplay Function

```
function frect(fountain, particle) {
   fill(fountain.f.colors[int(particle.life*fountain.f.colors.length)]);
   noStroke();
   rect(particle.x, particle.y, particle.partSize, particle.partSize);
}

var fdisplay = {ellipse: fellipse, rect: frect, image: fimage};
```

Definitions of Particle and Fountain and a JSON Fountain String
http://jsfiddle.net/bobcook/2kfbmwtu/gz

Projectiles

The second utility class (Projectile) was designed for weapons of all types, which covers a lot of ground. There had to lots of flexibility in bullet and projectile shapes. Luckily, flexibility is JavaScript's best feature. The design allows the user to define guns and their properties external to the class and to draw projectiles on demand external to the class. Further, projectiles must be physics bodies since they are required to have an impact upon collision.

The JSON evaluator for projectiles supports all of the Fountain properties except "sizePercent", which should be set to zero. New properties have been added for weapons and the physics of projectiles.

Each weapon has a **load**, which is the number of bullets per shot. The **angle** of a shot is the random spread in degrees around the shooting point. If the projectile load is one, the spread has a random effect on each shot. The **period** of a gun is the time that must elapse in milliseconds before it can be fired again. All bullets are particles so the **speed** property affects the velocity.

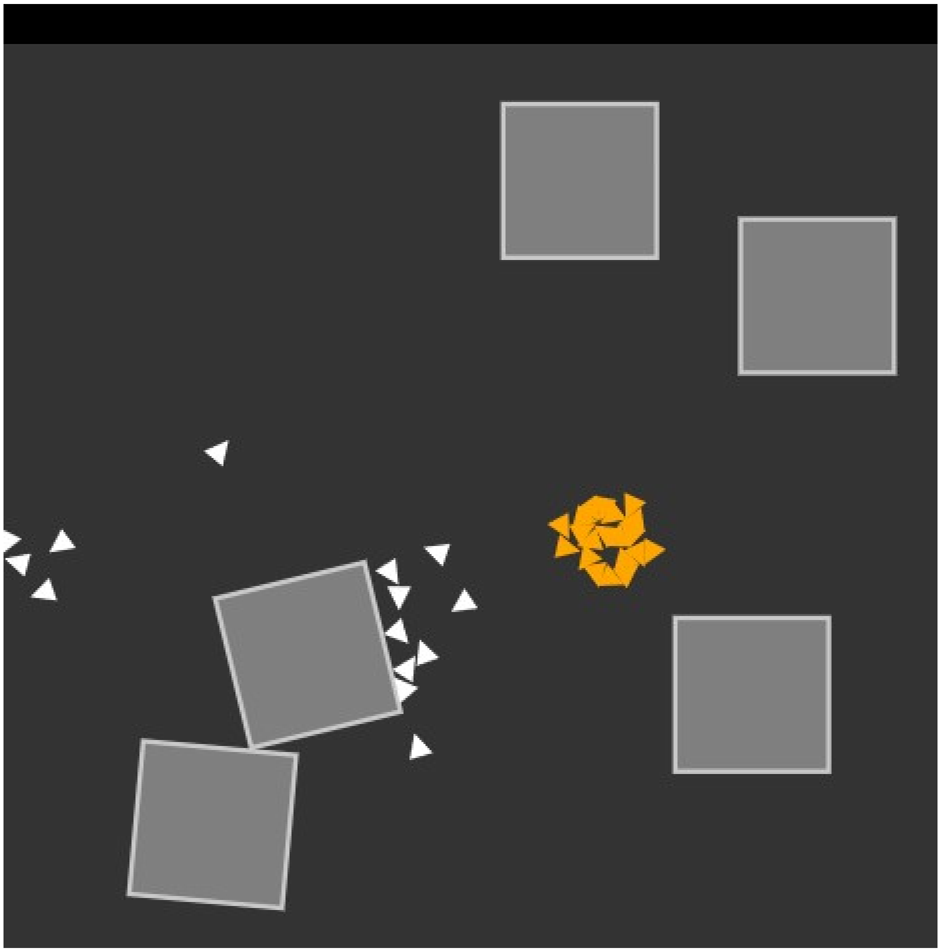
Remember that particles are not physics bodies. There are so many particles that modelling collisions would not really contribute anything to the "effect".

Bullets, however, are designed to hit things and to have an impact. Thus, all bullets are physics bodies. The following table lists the added JSON Projectile properties.

JSON Projectile Properties (in addition to Fountain properties)

Name	**Description**	**Example**
sizePercent	not used. Set to zero.	"sizePercent": 0
shape	string index into the fdisplay array to select a drawing function. The function must define the parameters body, shape, and position since it is a physics callback. Use the Fountain "file" property for images.	"shape": "pfill"
gravity	sets the gravityScale of the physics body.	"gravity": 1
angle	defines the spread of the projectile stream in degrees around the velocity vector.	"angle": [-4, 4]
load	number of bullets per shot.	"load": 5
period	number of milliseconds after a shot to reload.	"period": 4000
pshape	physics shape: box, circle or polygon.	"pshape": "polygon"
Optional		
density	density of the physics body.	"density": 1
friction	friction of the physics body.	"friction": 0.2
bounce	restitution of the physics body.	"bounce": 0.5
collidesWith	categories-to-collide-with string.	"collidesWith": "2+1"
categories	collision categories string.	"categories": "8+4+2"

JSON Example
http://jsfiddle.net/bobcook/qw9gw1cu/



The Projectile example creates five physics box-bodies at random screen locations. When the mouse is clicked, twenty triangular projectiles are fired toward that location. Observe that the bullets have mass so when they hit a box, it moves proportional to the ratio of their masses. Any image can be defined as a projectile, i.e. flowers, ducks, bricks, etc. Also note that while the graphics is impressive, the code for the program is very simple.

```
var defs;
var of;

function setup() {
   createCanvas(600, 400);
   b2newWorld(30, createVector(0, 0));
   for (var i=0; i<5; i++)
     var b = new b2Body('box', true,
        createVector(random(50,550),random(50,350)),
createVector(64,64));
   var t =
      '{   ' +
      '    "parts": [   ' +
      '    {   ' +
      '    "name": "foo",   ' +
```

```
function draw() {
  background(51);
  b2Update();
  b2Draw(false);
  of.Step();
  noStroke();
  text(of.length, width/2, 20);
  stroke(0);

}

//fire a shot toward the click
location
function mousePressed() {
   of.Create(mouseX, mouseY);
```

```
      '   "color":   ' +
      '      ["orange",[255,0,0],"yellow","white","white","white"],   ' +
      '   "shape": "pfill",   ' +
      '   "pshape": "polygon",   ' +
      '   "edges":  [[5,-5],[-5,-5],[0,5]],   '+
      '   "gravity": 1,   ' +
      '   "sizePercent": 0,   ' +
      '   "angle": [-4,4],   ' +
      '   "speed": 15.5,   ' +
      '   "limit": 99999,   ' +
      '   "lifetime": 60,   ' +
      '   "size": [4,8],   ' +
      '   "period": 1000,   ' +
      '   "load": 20,   ' +
      '   "rotation": 6,   ' +
      '   "x": 2,   ' +
      '   "y": 2}]}';
   defs = JSON.parse(t);
   of = new Projectile(defs, 'foo');
}
```

```
}
```

Projectile Class Definition (inherits from Fountain)

Methods	
new Projectile(JSON table, name, x, y)	xy are optional and specify the pixel origin for the shooter. If omitted, JSON xy are used. Both Fountain and Projectile properties are applied.
Create(x, y)	xy specify the location at which to fire. The shot will not take place until the weapon is re-armed.
Draw()	does not apply. The physics code controls drawing.
Step()	If Create fired a shot, the action occurs here, but only if the weapon is armed.
Stop()	does not apply. A shot cannot be stopped.
Properties (includes Fountain)	
fire	null or angle of fire from Create.
pedges	edge array converted to P5 vectors.
New Particle Property	
body	reference to the physics body.
New b2Body Properties	
fountain	references Fountain instance, used for drawing properties..

particle	references Particle instance, used in draw function.

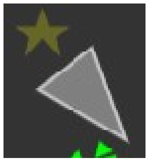
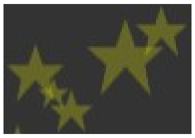
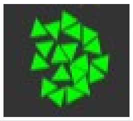
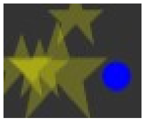
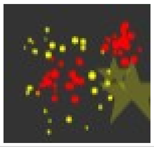
Sample fdisplay Function

```
function ffill(body, shape, position) {
   fill(body.fountain.f.colors[
      Math.floor(body.particle.life*body.fountain.f.colors.length)]);
   noStroke();
   b2Display(body, shape, position);
}

var fdisplay = {pfill: ffill, ellipse: fellipse, rect: frect, image: fimage};
```

Finally, we discuss the implementation of the main program (using step-wise refinement, of course!). The first step is to identify the game components, then they can be implemented one step at a time.

Try to implement each step yourself before peeking at the Fiddle listing. (Expect many sessions with the FireFox debugger on any project.)

Game Components

Alien Ship	A long triangle in the Fiddle, but an image in the PC version. Flies from a random left-side point towards a random right-side point. Should point in the direction of flight initially. Once a ship is destroyed, create another.	
Score	Displayed in the top-middle of the screen	Score: 700
Star Field	Move right-to-left and increases in size to enhance motion and distance "feel". Created as a Fountain.	
Multiple Shot	From earlier example. Shotgun projectile with 20 triangular bullets.	
Shield Buster	Blue phason circle that penetrates a shield and destroys a ship. Requires collision detection with the ship, but not other bullets.	
Explosion	Created as a Fountain at the point of ship destruction.	

Score

The "score" variable is declared globally so that it can be accessed from any function. The score display is drawn using a text font. It took a bit of experimenting to choose a font and font size.

```
var score = 0;
function draw() {
   background(51);
   textFont("Arial");
   textSize(40);
   fill(255);
   text("Score: " + score, width / 2 - 100, 50);
   stroke(0);
}
```

Alien Ship

The alien ship is a triangular physics object. A triangle must be drawn as a three-sided polygon. Remember that the points must be listed in counter-clockwise order.

The "ship" variable is also declared globally. We set the variable to null to indicate that a ship needs to be created. When a ship is destroyed, the variable is set to null. The next code written creates the ship and then applies an impulse to move it horizontally from left to right.

```
var ship = null;
function draw() {
   background(51);
   if (ship == null) {
     var xy = createVector(2, width/2);
     var target = createVector(1, 0); //positive X direction
     var angle = 0;
     ship = new b2Body('polygon', true, xy, [createVector(0, 15), createVector(20, -40), createVector(-20,
-40)],
       1, 0.5, 0.2, angle);
     ship.applyImpulse(target, random(2, 8));
   }
}
```

Continuing, we need to add code to check if the ship has been knocked off the screen (+100) or if it has crossed the right edge (-100). The code was tested by launching the ship in different directions. The first bug was encountered at this point because the code did not handle the (> height) case. The physics engine deactivates a body if it leaves the screen. The bug is fixed in the following code.

```
var ship = null;
function draw() {
   background(51);
   if (ship == null) {
     var xy = createVector(2, width/2);
     var target = createVector(1, 0);
     var angle = 0;
     ship = new b2Body('polygon', true, xy, [createVector(0, 15), createVector(20, -40), createVector(-20,
-40)],
    1, 0.5, 0.2, angle);
   ship.applyImpulse(target, random(2, 8));
  } else {
    var xy = ship.xy;
    if (xy.x < 0 || xy.y < 0) {
      score += 100;
      ship.destroy();
      ship = null;
    } else if (xy.x > width) {
      score -= 100;
      ship.destroy();
      ship = null;
    } else ship.applyForce(ship.velocity, 0.03);
  }
  b2Update();
  if (ship != null && !ship.active) { //off bottom case
    score += 100;
    ship = null;
  }
  b2Draw(false);
}
```

The remaining challenge is to fly the ship from a random left point to a random right point and to rotate the ship towards the target location. Given the two points, we need the calculate the velocity vector in order to apply the impulse properly. Further, given the (0, 1) initial orientation of the ship, the angle to rotate the nose must be calculated.

The velocity vector is just the vector-difference between the two points. However, the separation is hundreds of pixels, which is too large to use. Therefore, the result must be vector-normalized. Check out the Wikipedia article on Math.atan2. The function calculates the angle assuming that the gun is horizontal to start. In our case, the triangle needs to be rotated -90 degrees, or -PI/2 radians, to be horizontal.

```
var xy = createVector(2, random(height-30) + 10);
var target = createVector(width, random(height-30) + 10);
target.sub(xy);
var angle = -PI/2 + Math.atan2(target.y, target.x);
```

Star Field

The star field is created as a Fountain. I made the stars partially transparent to enhance the depth effect. I also created them small and then increased the size as they moved left. Because the definition was a JSON string, it was easy to experiment with different parameters. The hardest part was determining the points for a star shape. I found a star graphic on the Internet, downloaded it into a Paint program, enlarged the view, and then copied down the end points in a counter-clockwise direction. Remember that the "shape" name refers to a user-defined draw function in the "fdisplay" table.

The challenge in coding the star field was to generate the stars across the right side of the screen, not at a single point. Remember that the Fountain.Create function defines optional xy parameters. Aha!

```
st.Step();
st.Create(width-1, random(0,height));
st.Draw();
```

```
'    "name": "stars",    ' +
'    "color":    ' +
'        [[255,255,0,64]],    ' +
'    "shape": "stars",    ' +
'    "gravity": 0,    ' +
'    "sizePercent": 1.01,    ' +
'    "angle": [180, 180],    ' +
'    "speed": 10,    ' +
'    "limit": 25000000,    ' +
'    "lifetime": 260,    ' +
'    "size": [0.05, 0.4],    ' +
'    "x": 2,    ' +
'    "y": 2}    ' +
```

```
function fstars(fountain, particle) {
   push();
   noStroke();
   fill(fountain.f.colors[Math.floor(particle.life * fountain.f.colors.length)]);
   translate(particle.x, particle.y);
   scale(particle.partSize);
   beginShape();  //points are relative to the center
   vertex(32, -8);
   vertex(8, -8);
   vertex(0, -32);
   vertex(-8, -8);
   vertex(-32, -8);
   vertex(-16, 7);
   vertex(-21, 32);
   vertex(0, 8);
   vertex(21, 32);
   vertex(16, 7);
   endShape(CLOSE);
   pop();
}
```

```
var fdisplay = {
   stars: fstars,
   pfill: ffill,
   ellipse: fellipse,
   rect: frect,
   image: fimage
};
```

Multiple Shot

Implement multiple shot projectiles only required copying the JSON definition from the Projectile test code. That code fired a shot in the direction of a mouse click so no change was needed there. However, there was a mysterious bug. Color names stopped working when using the physics library, so we just set the color to green by using the array notation [0,255,0]. We also had the set the "bullet" property to true as sometimes shots at an angle would "miss" collision detection.

Observe that the shots are powerful enough to knock a ship off course, but not powerful enough to penetrate the shields. At this point, we discovered that it was hard to beat the game!

Shield Buster

We added a "shield buster" projectile to give the player an additional advantage and to illustrate collision handling. Remember that the default collision category is one. A new category "4" was defined for the bullets and their "collidesWith" property was set to "4". Then we had to change the "categories" property of the ship to "4" and set its "collidesWith to "4+1". We also attached a "collision" function to the ship. These bullets are also fired towards a mouse click, but from a different location.

```
function collider(bodyA, bodyB) {
   //test for ship and shield buster
   if (bodyA.categories == 4 && bodyB.categories == 4 && ship != null) {
      var xy = ship.xy;
      var x = new Fountain(defs, 'explosion', xy.x, xy.y);
      ex.push(x);
      ship.destroy();
      score += 100;
      ship = null;
   }
}

ship.categories = 4;
ship.collidesWith = 4 + 1;
ship.collision = collider;

'   "name": "shieldBuster",   ' +
'   "color":   ' +
'      [[0,0,255]],   ' +
'   "shape": "pfill",   ' +
'   "pshape": "circle",   ' +
'   "gravity": 1,   ' +
'   "sizePercent": 0,   ' +
'   "angle": [0,0],   ' +
'   "speed": 50,   ' +
'   "limit": 99999,   ' +
'   "lifetime": 80,   ' +
'   "size": [16,16],   ' +
'   "period": 2000,   ' +
'   "load": 1,   ' +
'   "bullet": true,   ' +
'   "categories": "4",   ' +
'   "collidesWith": "4",   ' +
'   "x": 1,   ' +
'   "y": 4},  ' +
```

Explosion

The explosion that occurs upon ship destruction is also defined as a Fountain. I had to experiment with the parameters to get a reasonable "look". The challenge with this final component was that multiple explosions could be active simultaneously. As a result, the code had to create particles for all active explosions and to delete explosions that had terminated.

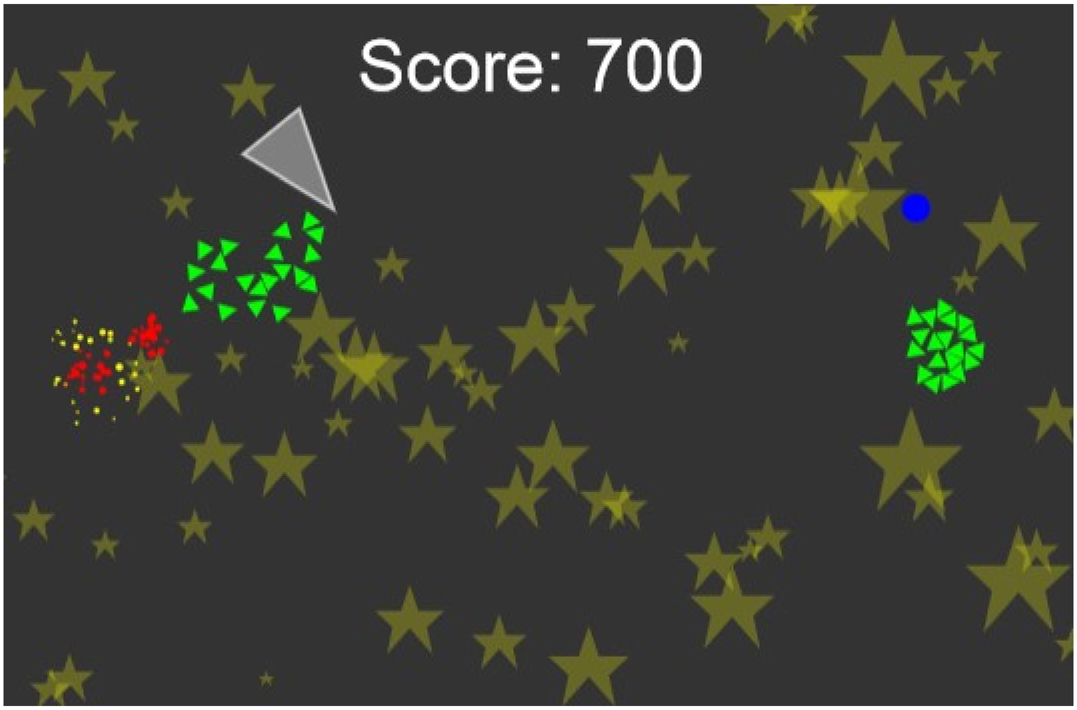
The solution was to declare an array "ex" to store active explosions. Remember that the Fountain class defines a "done" property, which can be checked to determine a Fountain's completion status. Also, remember that the Fountain constructor has optional xy parameters that can be set to locate particle streams.

```
'   "name": "explosion",   ' +
'   "color":   ' +
'      [[255,0,0],[255,255,0]],   ' +
'   "shape": "ellipse",   ' +
'   "gravity": 0,   ' +
'   "sizePercent": 0.99,   ' +
'   "angle": [0,360],   ' +
'   "speed": 0.5,   ' +
'   "limit": 250,   ' +
'   "lifetime": 60,   ' +
'   "size": [2,6],   ' +
'   "x": 2,   ' +
'   "y": 2},  ' +
```

```
var ex = [];

for (var i = 0; i < ex.length; i++) {
        ex[i].Step();
        ex[i].Create();
        ex[i].Draw();
        if (ex[i].done) ex.splice(i, 1);
   }
```

Final Result

Alien Attack
http://jsfiddle.net/bobcook/ps657k6t/


Score: 700