Chapter 5. Physics Libraries

"A library implies an act of faith/Which generations still in darkness hid/Sign in their night in witness of the dawn."
-- Victor Hugo

In this Chapter:

- To library or not to library
- A Box2D world
  - Bodies, Shapes, Joints
  - ContactListener
  - Interaction
- Toxiclibs
  - Verlet Physics
  - Connected Systems
  - "Behaviors"

5.1 To library or not to library?

Let's revisit some of the things we've done in the first four chapters.

1) Learn about concepts from the world of physics — What is a vector? What is a force? What is a wave? etc.

2) Understand the math and algorithms behind such concepts.

3) Implement the algorithms in Processing with an object-oriented approach.

These activities have yielded a set of motion simulation examples, allowing us to creatively define the physics of the worlds we build (whether realistic or fantastical). Of course, we aren't the first to try this. The world of computer graphics and programming is full of source code dedicated to simulation. Just try Googling "open-source physics engine" and you could spend the rest of your day pouring through rich and complex code. And so we must ask the question: If a code library will take care of physics simulation, why should we bother learning how to write any of the algorithms ourselves?

Here is where the philosophy behind this book comes into play. While many of the libraries out there give us physics (and super awesome advanced physics at that) for free, there are significant reasons for learning the fundamentals from scratch before diving into libraries. First, without an understanding of vectors, forces, and trigonometry, we'd be completely lost just reading the documentation of a library. Second, even though a library may take care of the math for us, it won't necessarily simplify our code. As we'll see in a moment, there can be a great deal of overhead in simply understanding how a library works and what it expects from you code-wise. Finally, as wonderful as a physics engine might be, if you look deep down into your hearts, it's likely that you seek to create worlds and visualizations that stretch the limits of imagination. A

library is great, but it provides a limited set of features. It's important to know both when to live within limitations in the pursuit of a Processing project and when those limits prove to be confining.

This chapter is dedicated to examining two open-source physics libraries—Box2D and toxiclibs *VerletPhysics* engine. With each library, we'll evaluate its pros and cons and look at reasons why you might choose one of these libraries for a given project.

5.2 What is Box2D and when might I use it?

Box2D began as a set of physics tutorials written in C++ by Erin Catto for the Game Developer's Conference in 2006. Over the last five years it has evolved into an elaborate and rich open-source physics engine. It's been used for countless projects, most notably highly successful games such as the award-winning puzzle game Crayon Physics and the runaway mobile and tablet hit Angry Birds.

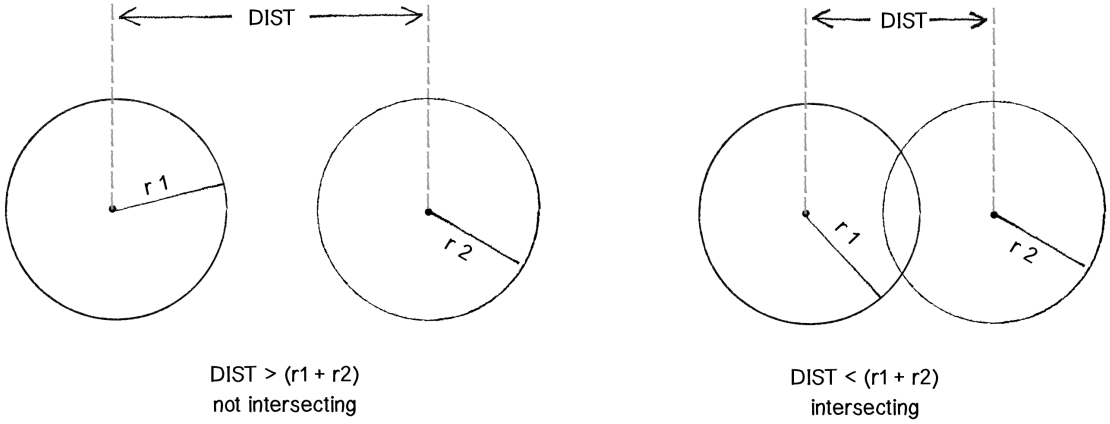
One of the key things to realize about Box2D is that it is a true physics engine. Box2D knows nothing about computer graphics and the world of pixels; it is simply a library that takes in numbers and spits out more numbers. And what are those numbers? Meters, kilograms, seconds, etc. All of Box2D's measurements and calculations are for real-world measurements, only its "world" is a two-dimensional plane with a top, bottom, left and right edge. You tell it things like: "The world has a gravitational force of 9.5 Newtons, and a circle with a radius of four meters and a mass of fifty kilograms is located ten meters above the world's bottom." Box2D will then tell you things like "One second later, the rectangle is at five meters from the bottom; two seconds later, it is ten meters below" etc. While this provides for an amazing and realistic physics engine, it also necessitates lots of complicated code in order to translate back and forth between the physics "world" (a key term in Box2D) and the world we want to draw on —the "pixel" world of Processing.

So when is it worth it to have this additional overhead? If I just want to simulate a circle falling down a Processing window with gravity, do I really need to write all the extra Box2D code just to get that effect? Certainly, the answer is no. We saw how to do this rather easily in just the first chapter of this book. Let's consider another scenario. What if I want to have a hundred of those circles falling? And what if those circles aren't circles at all—rather, irregularly shaped polygons? And what if I want these polygons to bounce off each other in a realistic manner when they collide?

You may have noticed that the first four chapters of this book, while covering motion and forces in detail, has skipped over a rather important aspect of physics simulation—*collisions*. Let's pretend for a moment that you aren't reading a chapter about libraries and that we decided right now to cover how to handle collisions in a particle system. We'd have to evaluate and learn two distinct algorithms that address these questions:

1. How do I determine if two shapes are colliding (i.e. intersecting)?
2. How do I determine the shapes' velocity after the collision?

If we're thinking about shapes like rectangles or circles, question #1 isn't too tough. You've likely encountered this before. For example, we know two circles are intersecting if the distance between them is less than the sum of their radii.



OK. Now that we know how to determine if two circles are colliding, how do we calculate their velocities after the collision? This is where we're going to stop our discussion. Why, you ask? It's not that understanding the math behind collisions isn't important or valuable (and because of this I'm including additional examples on the web site related to collisions without a physics library.) The reason for stopping is that life is short (let this also be a reason for you to consider going outside and frolicking instead of programming altogether). We can't expect to master every detail of physics simulation. And while we could continue this discussion for circles, it's only going to lead us to wanting to work with rectangles. And strangely shaped polygons. And curved surfaces. And swinging pendulums colliding with springy springs. And and and and and.

Working with collisions in our Processing sketch while still having time to spend with our friends and family, this is the reason for this chapter. Erin Catto spent years developing solutions to these kinds of problems, and at least for now, we don't need to engineer them ourselves.

In conclusion, if you find yourself describing an idea for a Processing sketch and the word "collisions" comes up, then it's likely time to learn Box2D. (We'll also encounter other words that might lead you down this path to Box2D, such as joint, hinge, pulley, motor, etc.)

5.3 How do I get Box2D in Processing?

So, if Box2D is a physics engine that knows nothing about pixel-based computer graphics *and* is written in C++, how are we supposed to use it in Processing?

The good news is that Box2D is such an amazing and useful library that everyone wants to use it —Flash, Javascript, Python, Ruby programmers. Oh, and Java programmers. There is something called JBox2D, a Java port of Box2D. And because Processing is built on top of Java, JBox2D can be used directly in Processing!

So here's where we are so far.

Box2D site for reference: http://www.box2d.org/
JBox2D site for Processing compatibility: http://www.jbox2d.org/

This is all you need to get started writing Box2D code in Processing. However, as we are going to see in a moment, there are several pieces of functionality we'll repeatedly need in our Processing code, and so it's worth having one additional layer between our sketches and JBox2D. I'm calling this PBox2D—a Processing Box2d "helper" library included as part of this book's code example downloads.

PBox2D: https://github.com/shiffman/PBox2D ***[temporary URL until book web site exists?]***

It's important to realize that PBox2D is not a Processing wrapper for all of Box2D. After all, Box2D is a thoughtfully organized and well-structured API and there's no reason to take it apart and re-implement. However, it's useful to have a small set of functions that help you get your Box2D world set up as well as help you to figure out where to draw your Box2D shapes. And this is what PBox2D will provide.

I should also mention before we move forward that there are also other Processing libraries that wrap Box2D for you, one I would recommend taking a look at is Fisica (http://www.ricardmarxer.com/fisica/) by Ricard Marxer.

5.4 Box2D Basics—Process

Do not despair! We really are going to get to the code very soon and in some ways blow the lid off our previous work. But before we're ready to do that, it's important to walk through the overall process of using Box2D in Processing. Let's begin by writing a pseudo-code generalization of all of our examples in chapters one through four.

SETUP:
1) Create all the objects in our world.

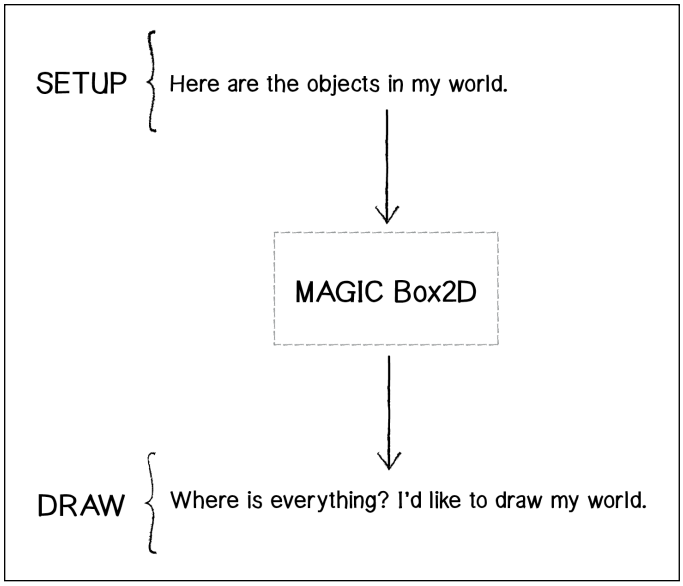
DRAW:

2) Calculate all the forces in our world.
3) Apply all the forces to our objects (F = M * A).
4) Update the locations of all the objects based on their acceleration.
5) Draw all of our objects.

Great. Let's rewrite this pseudo-code as it will appear in our Box2D examples.

SETUP:
1) Create all the objects in our world.

DRAW:
2) Draw all of our objects.



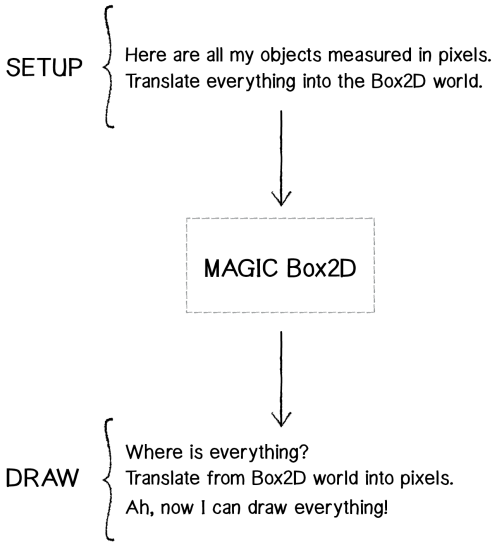
This, of course, is the fantasy of Box2D. We've eliminated all of those painful steps of figuring out how the objects are moving according to velocity and acceleration. Box2D is going to take care of this for us! The good news is that this does accurately reflect the overall process. Let's imagine Box2D as a magic box.

In ***setup()***, we're going to say to Box2D: "Hello there. Here are all of the things I want in my world." In draw(), we're going to politely ask Box2D: "Oh, hello again. If it's not too much trouble, I'd like to draw all of those things in my world. Could you tell me where they are?"

The bad news: it's not as simple as the above methodology would lead you to believe. For one, making the stuff that goes in the Box2D world involves wading through the documentation for how different kinds of shapes are built and configured. Second, we have to remember we can't tell Box2D anything about pixels, as it will simply get confused and fall apart. Before we tell Box2D what we want in our world, we have to convert our pixel units to Box2D "world" units. And the same is true when it comes time to draw our stuff. Box2D is going to tell us the location of the things in its world, which we then have to translate for the pixel world.

SETUP:
1) Create everything that lives in our pixel world.
2) Translate the pixel world into the Box2D world.

DRAW:
3) Ask Box2D where everything is.
4) Translate Box2D's answer into the pixel world.
5) Draw everything.



5.5 Box2D Basics—Core Elements

Now that we understand that anything we create in our Processing sketch has to be placed into the Box2D world, let's look at an overview of the elements that make up that world.

Core Elements of a Box2D World:

1. **World**: The Box2D "World" manages the physics simulation. It knows everything about the overall coordinate space as well as stores lists of every element in the world (see 2-4 below).
2. **Body:** A Box2D "Body" is the primary element in the Box2D world. It has a location. It has a velocity. Sound familiar? The Body is essentially the class we've been writing on our own in our vectors and forces examples.
3. **Shape:** A Box2D "Shape" keeps track of all the necessary collision geometry attached to a Body.
4. **Fixture:** A Box2D "Fixture" attaches a Shape to a Body and sets properties such as density, friction, and restitution.
5. **Joint:** A Box2D "Joint" is a connection between two bodies (or between one body and the world itself).

In the next four sections, we are going to walk through each of the above elements in detail, building several examples along the way. But before we are ready to do so there is one other important element we should briefly discuss.

6. **Vec2:** A Box2D "Vec2" describes a vector in the Box2D world.

And so here we are, arriving with trepidation at an unfortunate truth in the world of using physics libraries. Any physics simulation is going to involve the concept of a vector. This is the good part. After all, we just spent several chapters familiarizing ourselves with what it means to describe motion and forces with vectors. We don't have to learn anything new conceptually.

Now the part that makes the single tear fall from my eye: we don't get to use PVector. It's nice that Processing has PVector for us, but anytime you use a physics library you will probably discover that the library includes its own vector implementation. This makes sense, after all; why should Box2D be expected to know about PVector? And in many cases, the physics engine will want to implement a vector class it in a specific way so that it is especially compatible with the rest of the library's code. So while we don't have to learn anything new conceptually, we do have to get used to some new naming conventions and syntax. Let's quickly demonstrate a few of the basics in ***Vec2*** as compared to those in ***PVector.***

Let's say we want to add two vectors together.

PVector	Vec2

`PVector a = new PVector(1,-1);`
`PVector b = new PVector(3,4);`
`a.add(b);`
	`Vec2 a = new Vec2(1,-1);`
	`Vec2 b = new Vec2(3,4);`
	`a.addLocal(b);`

`PVector a = new PVector(1,-1);`
`PVector b = new PVector(3,4);`
`PVector c = PVector.add(a,b);`
	`Vec2 a = new Vec2(1,-1);`
	`Vec2 b = new Vec2(3,4);`
	`Vec2 c =` **`a.add(b);`**

How about multiply/scale?

PVector	Vec2

`PVector a = new PVector(1,-1);`
`float n = 5;`
`a.mult(n);`
	`Vec2 a = new Vec2(1,-1);`
	`float n = 5;`
	`a.mulLocal(n);`

`PVector a = new PVector(1,-1);`
`float n = 5;`
`PVector c = PVector.mult(a,n);`
	`Vec2 a = new Vec2(1,-1);`
	`float n = 5;`
	`Vec2 c =` **`a.mul(n);`**

Magnitude and normalize?

PVector	Vec2

`PVector a = new PVector(1,-1);`
`float m = a.mag();`
`a.normalize();`
	`Vec2 a = new Vec2(1,-1);`
	`float m =` **`a.length();`**
	`a.normalize();`

As you can see, the concepts are the same, but the function names and the arguments are slightly different. For example, instead of static and non-static ***add()*** and ***mult()***, if a Vec2 is altered, the word "local" is included in the function name—***addLocal(), multLocal()***.

We'll cover what you need to know here, but if you are looking for more, full documentation of Vec2 can be found by downloading the JBox2D source at http://code.google.com/p/jbox2d/.

5.6 Living in a Box2D World

The Box2D "World" object is in charge of everything. It manages the coordinate space of the world, all of the stuff that lives in the world, and decides when time moves forward in the world.

In order to have Box2D as part of our Processing sketches, the "World" is the very first thing that needs to be set up. Here is where PBox2D comes in handy and takes care of making the world for us.

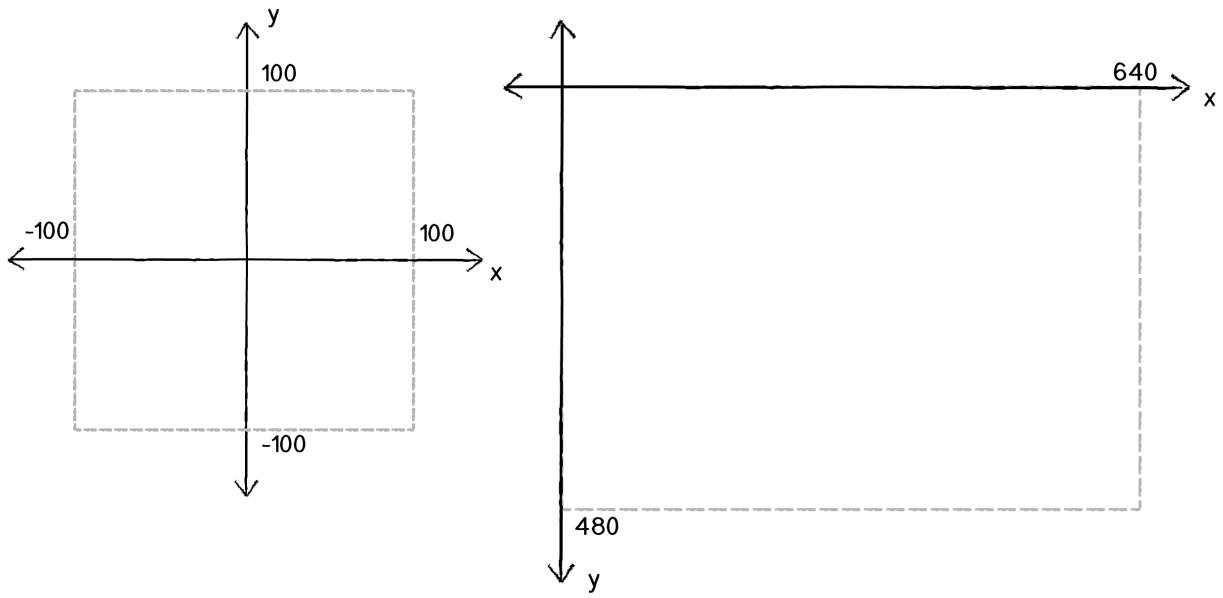
```
PBox2D box2d;

void setup() {
  box2d = new PBox2D(this);
  box2d.createWorld();            $$ Initializes a Box2D world with default settings
}
```

When you call ***createWorld()***, PBox2D will set up a default gravity for you (pointing down), however, you can always alter the gravity of your world by saying:

```
  box2d.setGravity(0, -10);
```

It's worth noting that gravity doesn't have to be fixed nor does it always have to point downwards; you can adjust the gravity vector while your program is running. Gravity can be turned off by setting it to a (0,0) vector.

So, what are those numbers 0 and -10? This should remind us of one of the most important details of using Box2D: the Box2D coordinate system is not your pixel coordinate system!! Let's look at how Box2D and a Processing window think differently of their worlds.



[DIAGRAM NEEDS LABELS, LEFT "BOX2D WORLD", RIGHT "PROCESSING WINDOW"]

Notice how in Box2D (0,0) is in the center and up is the positive direction along the y-axis! Box2D's coordinate system is just like that lovely old-fashioned Cartesian coordinate system you might have learned about in a high school geometry class. Processing, on the other hand, uses a traditional computer graphics coordinate system where (0,0) is in the top left corner and down is the positive direction along the y-axis. This is why if we want gravity to point down, we need to give Box2D a vector with a negative y-value.

```
Vec2 gravity = new Vec2(0, -10);
```

Luckily for us, if we prefer to think in terms of pixel coordinates (which as Processing programmers, we are likely to do), PBox2D offers a series of helper functions that convert between pixel space and Box2D space. Before we move onto the next section and look at creating Box2D bodies, let's take a look at how these helper functions work.

Let's say we want to tell Box2D where the mouse is in its world. We know the mouse is located at (mouseX,mouseY) in Processing. To convert it we say we want to convert a "coordinate" from "pixels" to "world"—***coordPixelsToWorld()***. Or:

```
Vec2 mouseWorld = box2d.coordPixelsToWorld(mouseX,mouseY);  $$ Convert mouseX,mouseY to
                                                            coordinate in Box2D world
```

What if we had a Box2D world coordinate and wanted to translate it to our pixel space?

```
Vec2 worldPos = new Vec2(-10,25);        $$ To demonstrate, let's just make up a
                                            world position

Vec2 pixelPos = box2d.coordWorldToPixels(worldPos); $$ Convert to pixel space
ellipse(pixelPos.x, pixelPos.y,16,16);              This is necessary because ultimately we
                                                    are going to want to draw the elements in
                                                    our window
```

PBox2D has a set of functions to take care of translating back and forth between the Box2D world and pixels. It's probably easier to learn about all of these functions during the course of actually implementing our examples, but let's quickly look over the list of the possibilities.

TASK	FUNCTION
Convert location from World to Pixels	***Vec2 coordWorldToPixels(Vec2 world)***
Convert location from World to Pixels	***Vec2 coordWorldToPixels(float worldX, float worldY)***
Convert location from Pixels to World	***Vec2 coordPixelsToWorld(Vec2 screen)***
Convert location from Pixels to World	***Vec2 coordPixelsToWorld(float pixelX, float pixelY)***

TASK	FUNCTION
Scale a dimension (such as height, width, or radius) from Pixels to World	***float scalarPixelsToWorld(float val)***
Scale a dimension from World to Pixels	***float scalarWorldToPixels(float val)***
Scale a vector from Pixels To World	***Vec2 vectorPixelsToWorld(Vec2 v)***
Scale a vector from World To Pixels	***Vec2 vectorWorldToPixels(Vec2 v)***

There are also additional functions that allow you to pass or receive a PVector when translating back and forth, but since we are only working with Box2D in the examples in this chapter, it's easiest to stick with the Vec2 class for all vectors.

Once the world is initialized, we are ready to actually put stuff in the world—Box2D bodies.

5.6 Building a Box2D Body.

A Box2D body is the primary element in the Box2D world. It's the equivalent to the "Mover" class we built on our own in previous chapters, it is the thing that moves around the space and experiences forces. It can also be static (meaning fixed and not moving). It's important to note, however, that a Body has no geometry, it isn't anything physically. Rather, bodies have Box2D Shapes attached to them (this way a Body can be a single rectangle or a rectangle attached to a circle, etc.) We'll look at Shapes in a moment, for now let's see how we first build a Body.

Step 1. Define a Body.

The first thing we have to do is create a "Body Definition." This will let us define the properties of the Body we intend to make. This may seem a bit awkward at first, but this is how Box2D is structured. Anytime you want to make a "thing" you have to make a "thing definition" first. This will hold true for bodies, shapes, and joints.

```
BodyDef bd = new BodyDef();  $$ Make a Body Definition before making a Body
```

Step 2. Configure the Body Definition.

The Body Definition is where we can set specific properties or attributes of the Body we intend to make. One attribute of a Body, for example, is its starting location. Let's say we want to position the Body in the center of the Processing window.

```
Vec2 center = new Vec2(width/2,height/2); $$ A Vec2 in the center of the Processing window
```

Danger, danger! I'm not going to address this with every single example, but it's important to at least point out the perilous path we are taking with the above line of code. Remember, if we are going to tell Box2D where we want the Body to start, we must give Box2D a world coordinate!

Yes, we want to think of its location in terms of pixels, but Box2D doesn't care. And so before we pass that position to the Body Definition, we must make sure to use one of our helper conversion functions.

```
Vec2 center = box2d.coordPixelsToWorld(width/2,height/2); $$ A Vec2 in the center of the
                                                            Processing window converted to Box2D
                                                           World coordinates!
bd.position.set(center);  $$ Setting the position attribute of the Box2D Body Definition
```

The Body definition must also specify the "type" of Body we want to make. There are three possibilities:

- ***Dynamic.*** This is what we will use most often, a "fully simulated" body. A dynamic body moves around the world, collides with other bodies, and responds to the forces in its environment.
- ***Static.*** A static body is one that cannot move (as if it had an infinite mass). We'll use static bodies for fixed platforms and boundaries.
- ***Kinematic.*** A kinematic body can be moved manually by setting its velocity directly. If you have a user-controlled object in your world, you can use a kinematic body. Note that kinematic bodies collide only with dynamic bodies and not other static or kinematic ones.

There are several other properties you can set in the Body definition. For example, if you want to body to have a fixed rotation (i.e. never rotate) you can say:

```
bd.fixedRotation = true;
```

You can also set a value for linear or angular damping so that the objects continously slows down as if there is friction).

```
bd.linearDamping = 0.8;
bd.angularDamping = 0.9;
```

In addition, fast moving objects in Box2D should be set as bullets. This tells the Box2D engine that the object may move very quickly and to check its collisions more carefully so that it doesn't by accident jump over another body.

```
bd.bullet = true;
```

Step 3. Create the Body

Once we're done with the definition (BodyDef), we can create the Body object itself. PBox2D provides a helper function for this—***createBody()***.

```
Body body = box2d.createBody(bd);  $$ The Body object is created by passing in the Body
                                      Definition. (This allows for making multiple bodies from
                                      one definition).
```

Step 4. Set any other conditions for the Body's starting state

Finally, though not required, if you want to set any other initial conditions for the Body, such as linear or angular velocity, you can do so with the newly created Body object.

```
body.setLinearVelocity(new Vec2(0,3));   $$ Setting an arbitrary initial velocity
body.setAngularVelocity(1.2);            $$ Setting an arbitrary initial angular velocity
```

5.7 Attaching a Box2D Shape to a Body with a Fixture.

A Body on its own doesn't physically exist in the world. It's like a soul with no human form to inhabit. For a Body to have mass, we must first define a Shape and attach that Shape to the Body with something known as a Fixture.

The job of a Box2D Shape is to keep track of all the necessary collision geometry attached to a Body. A Shape also has several important properties that affect the Body's motion. There is *density*, which ultimately determines that Body's mass. Shapes also have *friction* and *restitution* ("bounciness") which will be defined through a Fixture. One of the nice things about Box2D's methodology, which separates the concepts of Bodies and Shapes into two separate objects, is that you can attach multiple shapes to a single Body in order to create more complex forms. We'll see this in a future example.

To create a Shape, we need to first decide what kind of shape we want to make. For most non-circular shapes, a PolygonShape will work just fine. For example, let's look at how we define a rectangle.

Step 1. Define a Shape.

```
PolygonShape ps = new PolygonShape();  $$ Define the shape: a polygon
```

Next up, we have to define the width and height of the rectangle. Let's say we want our rectangle to be 150×100 pixels. Remember, pixel units are no good for Box2D shapes! So we have to use our helper functions to convert them first.

```
float box2Dw = box2d.scalarPixelsToWorld(150);  $$ Scale dimensions from pixels to Box2D world
float box2Dh = box2d.scalarPixelsToWorld(100);

ps.setAsBox(box2Dw, box2Dh);         $$ Use setAsBox() function to define shape as a rectangle
```

Step 2. Create a Fixture.

The Shape and and Body are made as two separate entities. In order to attach a Shape to a Body, we must make a Fixture. A fixture is created, just as with the Body, via a FixtureDef (i.e. Fixture definition) and assigned a Shape.

```
FixtureDef fd = new FixtureDef();
fd.shape = ps;                    $$ The fixture is assigned the PolygonShape we just made
```

Once we have the Fixture Definition, we can set parameters that affect the physics for the Shape being attached.

```
fd.friction = 0.3;    $$ The coefficient of friction for the shape, typically between 0 and 1
fd.restitution = 0.5; $$ The Shape's restitution (i.e. elasticity), typically between 0 and 1
fd.density = 1.0;     $$ The Shape's density, measured in kilograms per meter squared.
```

Step 3. Attach the Shape to the Body with the Fixture.

Once the Fixture is defined, all we have left to do is attach the Shape to the Body with the Fixture by calling the ***createFixture()*** function.

```
body.createFixture(fd);      $$ Creates the Fixture and attaches the Shape to the Body object
```

I should note that Step 2 can be skipped if you do not need to set the physics properties (Box2D will use default values). You can create a Fixture and attach the Shape all in one step by saying:

```
body.createFixture(ps,1);    $$ Creates the Fixture and attaches the Shape with a density of 1
```

While most of our examples will take care of attaching Shapes only once when the Body is first built, this is not a limitation of Box2D. Box2D allows for Shapes to be created and destroyed on the fly.

Exercise: Knowing what you know about Box2D so far, fill in the blank in the code below that demonstrates how to make a circular shape in Box2D.

```
CircleShape cs = new CircleShape();
float radius = 10;
cs.m_radius = _____;
FixtureDef fd = new FixtureDef();
fd.shape = cs;
fd.density = 1;
fd.friction = 0.1;
fd.restitution = 0.3;

body.createFixture(fd);
```

5.8 Three's Company: Bodies and Shapes and Fixtures

Before we put any of this code we've been writing into a Processing sketch, let's review all the steps we took to construct a Body.

1. Define a Body using BodyDef (set any properties, such as location).
2. Create the Body from the Body Definition.

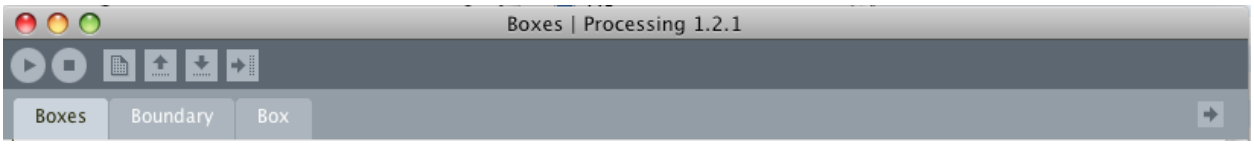
3. Define a Shape using PolygonShape, CircleShape, or any other Shape class.
4. Define a Fixture using FixtureDef and assign the Fixture a Shape (set any properties, such as friction, density, and restitution).
5. Attach the Shape to the Body

```
BodyDef bd = new BodyDef();                                    $$ STEP 1. Define the Body.
bd.position.set(box2d.coordPixelsToWorld(width/2,height/2));

Body body = box2d.createBody(bd);                              $$ Step 2. Create the Body.

PolygonShape ps = new PolygonShape();                          $$ Step 3. Define the Shape.
float w = box2d.scalarPixelsToWorld(150);
float h = box2d.scalarPixelsToWorld(100);
ps.setAsBox(w, h);


FixtureDef fd = new FixtureDef();                              $$ Step 4. Define the Fixture
fd.shape = ps;
fd.density = 1;
fd.friction = 0.3;
fd.restitution = 0.5;

body.createFixture(fd);                                $$ Step 5. Attach Shape to Body with Fixture
```

5.8 Box2D and Processing, reunited and it feels so good.

Once a Body is made it lives in the Box2D physics world. Box2D will always know it's there, check it for collisions, move it appropriately according to the forces, etc. It'll do all that for you without you having to lift a finger! What it won't do, however, is display the Body for you. This is a good thing. This is your time to shine. When working with Box2D what we're essentially saying is, "I want to be the designer of my world, and I want you, Box2D, to compute all the physics."

Now, Box2D will keep a list of all the Bodies that exist in the world. This can be accessed by calling the World object's getBodyList() function. Nevertheless, what I'm going to demonstrate here is a technique for keeping your own Body lists. Yes, this may be a bit redundant and we perhaps sacrifice a bit of efficiency. But we more than make up for that with ease of use. This methodology will allow us to program like we're used to in Processing, and we can easily keep track of which Bodies are which and render them appropriately. Let's consider the structure of the following Processing sketch:

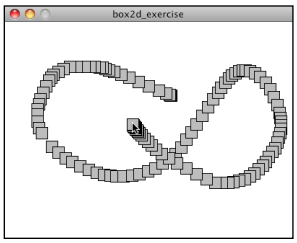


This looks like any ol' Processing sketch. We have a main tab called "Boxes" and a "Boundary" and "Box" tab. Let's think about the Box tab for a moment. The Box tab is where we will write a class to describe a Box object, a simple class to describe a rectangular body in our world.

```
class Box  {

  float x,y;                  $$ Our Box object has an x,y location and a width and a height
  float w,h;

  Box() {
    x = mouseX;               $$ Our Box object starts at the mouse location
    y = mouseY;
    w = 16;
    h = 16;
  }

  void display() {            $$ We draw the Box object using Processing's rect() function
    fill(175);
    stroke(0);
    rectMode(CENTER);
    rect(x,y,w,h);
  }
}
```

Let's write a main tab that creates a new Box whenever the mouse is pressed and stores all the Box objects in an ArrayList. (This is very similar to our approach in the Particle System examples from Chapter 4.)

Example: A comfotable and cozy Processing sketch that needs a little Box2D




```
ArrayList<Box> boxes;         $$ A list to store all Box objects

void setup() {
  size(400,300);
  boxes = new ArrayList<Box>();
}

void draw() {
  background(255);

  if (mousePressed) {                   $$ When the mouse is pressed, add a new Box object
    Box p = new Box(mouseX,mouseY);
    boxes.add(p);
  }

  for (Box b: boxes) {                  $$ Display all the Box objects
    b.display();
  }
}
```

Now, here's our assignment. Take the above example verbatim, but instead of drawing fixed boxes on the screen, draw boxes that experience physics (via Box2D) as soon as they appear.

We'll need two major steps to accomplish our goal.

Step 1. Add Box2D to our main program (i.e. setup() and draw())

This part is not too tough. We saw this already in our discussion of building a Box2D world. This is taken care of for us by the PBox2D helper class. We can create a PBox2D object and initialize it in ***setup()***.

```
PBox2D box2d;

void setup() {
  box2d = new PBox2D(this);  $$ Initialize and create the Box2D world
  box2d.createWorld();
}
```

Then in ***draw()***, we need to make sure we call one very important function: ***step()***. Without this function, nothing would ever happen! ***step()*** advances the Box2D world a step further in time. Internally, Box2D sweeps through and looks at all of the Bodies and figures out what to do with them. Just calling ***step()*** on its own moves the Box2D world forward with default settings; however, it is customizable (and this is documented in the PBox2D source).

```
void draw() {
  box2d.step();    $$ We must always step through time!
}
```

Step 2. Link every Processing Box object with a Box2D Body object

As of this moment, the Box class includes variables for location and width and height. What we now want to say is:

"I hereby relinquish the command of this object's position to Box2D. I no longer need to keep track of anything related to location, velocity, and acceleration. Instead, I only need to keep track of a Box2D body and have faith that Box2D will do the rest."

```
class Box  {

  Body body;      $$ Instead of any of the usual variables,
                    we will store a reference to a Box2D Body
  float w;
  float h;
```

We don't need (x,y) anymore since, as we'll see, the Body itself will keep track of its location. The Body technically could also keep track of the width and height for us, but since Box2D isn't going to do anything to alter those values over the life of the Box object, we might as well just hold onto them ourselves until it's time to draw the Box.

Then, in our constructor, in addition to initializing the width and height, we can go ahead and include all of the Body and Shape code we learned in the previous two sections!

```
  Box() {
    w = 16;
    h = 16;
```

```
    BodyDef bd = new BodyDef();                    $$ Build Body
    bd.position.set(box2d.coordPixelsToWorld(mouseX,mouseY));
    body = box2d.createBody(bd);


    PolygonShape ps = new PolygonShape();          $$ Build Shape
    float box2dW = box2d.scalarPixelsToWorld(w/2);
    float box2dH = box2d.scalarPixelsToWorld(h/2); $$ Box2D considers the width and height of a
    ps.setAsBox(box2dW, box2dH);                      rectangle to be the distance from the
                                                      center to the edge (so half of what we
    FixtureDef fd = new FixtureDef();                 normally think of as width or height.)
    fd.shape = ps;
    fd.density = 1;
    fd.friction = 0.3;                             $$ Set physics parameters
    fd.restitution = 0.5;

    body.createFixture(fd);                        $$ Attach Shape to Body with Fixture
  }
```

Ok, we're almost there. Before we introduced Box2D, it was easy to draw the Box. The object's location was stored in variables x and y.

```
  void display() {                $$ Drawing the object using rect()
    fill(175);
    stroke(0);
    rectMode(CENTER);
    rect(x,y,w,h);
  }
```

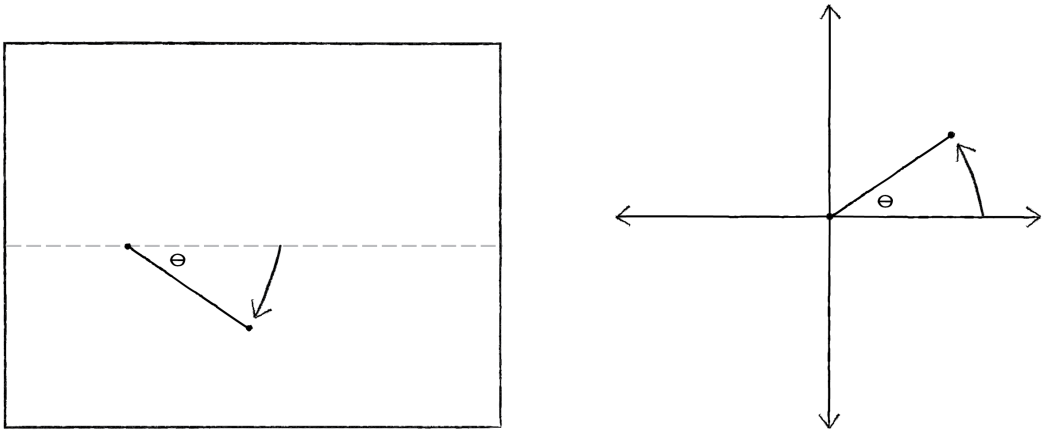
But now Box2D manages the object's motion. So we can no longer use our own variables to display the shape. But not to fear! Our Box object has a reference to the Box2D Body associated with it. So all we need to do is politely ask the Body,"Pardon me, where are you located?" Since this is a task we'll need to do quite often, PBox2D includes a helper function: ***getBodyPixelCoord()***.

```
Vec2 pos = box2d.getBodyPixelCoord(body);
```

Just knowing the location of a Body isn't enough; we also need to know its angle of rotation.

```
float a = body.getAngle();
```

Once we have the location and angle, it's easy to display the object using translate and rotate. Note, however, that the Box2D coordinate system considers rotation in the opposite direction from Processing, so we need to multiply the angle by -1.



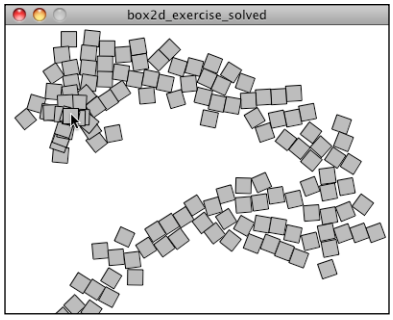
clockwise rotation in Processing

counter-clockwise rotation in Box2D

```
void display() {
  Vec2 pos = box2d.getBodyPixelCoord(body);        $$ We need the Body's location and angle
  float a = body.getAngle();

  pushMatrix();
  translate(pos.x,pos.y);        $$ Using the Vec2 position and float angle to
  rotate(-a);                       translate and rotate the rectangle
  fill(175);
  stroke(0);
  rectMode(CENTER);
  rect(0,0,w,h);
  popMatrix();
}
```

In case we want to have objects that can be removed from the Box2D world, it's also useful to include a function to destroy a Body, such as:

```
void killBody() {                $$ This function removes a Body from the Box2D world
  box2d.destroyBody(body);
}
```

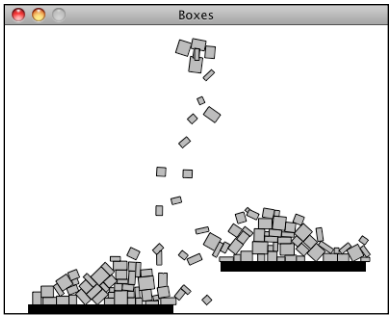


Exercise: In the code downloads for this chapter, find the sketch named "box2d_exercise." Using the methodology outlined in this chapter, add the necessary code to the main and Box tabs to implement Box2D physics. The result should appear as in the screenshot to the left. Be more creative in how you render the boxes.

5.9 Fixed Box2D objects

In the example we just created, the Box objects appear at the mouse location and fall downwards due to Box2D's default gravity force. What if we wanted to install some immovable boundaries in the Box2D world that would block the path of the Box objects (as in the illustration below)?

Box2D makes this easy for us by providing a means to lock a Body (and any associated Shapes) in place. Just set the BodyDef type to STATIC.




```
BodyDef bd = new BodyDef();
bd.type = BodyType.STATIC;    $$ When BodyDef type = STATIC, the
                                 Body is locked in place
```

We can add this feature to our Boxes example by writing a class called "Boundary" and having each Boundary object create a fixed Box2D body.

Example: Falling Boxes Hitting Boundaries

```
class Boundary {

  float x,y;                  $$ A boundary is a simple rectangle with x,y,width,and height
  float w,h;
  Body b;

  Boundary(float x_,float y_, float w_, float h_) {
    x = x_;
    y = y_;
    w = w_;
    h = h_;

    BodyDef bd = new BodyDef();                      $$ Build the Box2D Body and Shape
    bd.position.set(box2d.coordPixelsToWorld(x,y));
    bd.type = BodyType.STATIC;                       $$ Make it fixed by setting type to STATIC!
    b = box2d.createBody(bd);

    float box2dW = box2d.scalarPixelsToWorld(w/2);
    float box2dH = box2d.scalarPixelsToWorld(h/2);
    PolygonShape ps = new PolygonShape();

    b.createFixture(ps,1);                           $$ Using the createFixture() shortcut
  }

  void display() {                        $$ Since we know it can never move, we can just draw it
    fill(0);                                 the old-fashioned way, using our original variables
    stroke(0);                               No need to query Box2D.
    rectMode(CENTER);
    rect(x,y,w,h);
  }

}
```

5.9 A Curvy Boundary

If you want a fixed boundary that is a curved surface (as opposed to a polygon), this can be achieved with the Shape ***ChainShape.***

The ChainShape is another Shape like PolygonShape or CircleShape, so to include one in our system, we follow the same steps.

1. Define a Body

```
BodyDef bd = new BodyDef();
Body body = box2d.world.createBody(bd);
```

```
$$ The Body does not need a position; the
   EdgeShape will take care of that for us.
   It also does not need a type as it is STATIC
   by default.
```

2. Define the Shape

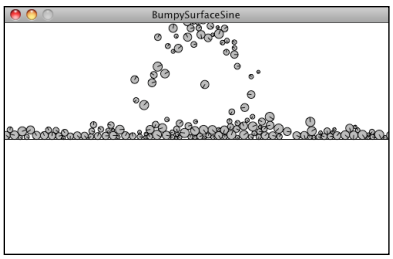
```
ChainShape chain = new ChainShape();
```

3. Configure the Shape

The ChainShape is a series of connected vertices. To create the chain, we must first specify an array of vertices (each as a Vec2 object). For example, if we wanted a straight line from the left-hand side of our window to the right-hand side, we would just need an array of two vertices: (0,150) and (width,150). (Note that if you want to create a loop where the first vertex connects to the last vertex in a loop you can use the ChainLoop class instead.)

```
Vec2[] vertices = new Vec2[2];       $$ Adding a vertex on the right side of window
vertices[0] = box2d.coordPixelsToWorld(0,150);
vertices[1] = box2d.coordPixelsToWorld(width,150);
                                     $$ Adding a vertex on the left side of window
```

To create the chain with the vertices, the array is then passed into a function called ***createChain()***.



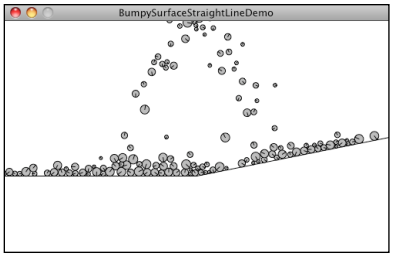

```
chain.createChain(vertices, vertices.length);
            $$ If you didn't want to use the entire array,
            you could specify a value less than length
```

4. Attach the Shape to the Body with a Fixture

A Shape is not part of Box2D unless it is attached to a Body. Even if it is a fixed boundary and never moves, it must still be attached. Just as with other shapes, the ChainShape can be given properties like restitution and friction with a Fixture.

```
FixtureDef fd = new FixtureDef();
fd.shape = chain;                  $$ A fixture assigned to the ChainShape
fd.density = 1;
fd.friction = 0.3;
fd.restitution = 0.5;

body.createFixture(fd);
```

Now, if we want to include an ChainShape in our sketch, we can follow the same strategy as we did with a fixed boundary. Let's write a class called Surface:



Example: EdgeChainDef with three hard-coded vertices

```
class Surface {
  ArrayList<Vec2> surface;

  Surface() {

    surface = new ArrayList<Vec2>();
    surface.add(new Vec2(0, height/2+50));          $$ 3 vertices in pixel coordinates
    surface.add(new Vec2(width/2, height/2+50));
    surface.add(new Vec2(width, height/2));

    ChainShape chain = new ChainShape();

    Vec2[] vertices = new Vec2[surface.size()];     $$ Make an array of Vec2 for the ChainShape
    for (int i = 0; i < vertices.length; i++) {
      vertices[i] = box2d.coordPixelsToWorld(surface.get(i));   $$ Convert each vertex to
    }                                                           Box2D World cooddinates

    chain.createChain(vertices, vertices.length);   $$ Create the ChainShape with array of Vec2

    BodyDef bd = new BodyDef();                     $$ Attach the Shape to the Body
    Body body = box2d.world.createBody(bd);

    body.createFixture(chain, 1);
  }
```

Notice how the above class includes an ArrayList to store a series of Vec2 objects. Even though we fully intend to store the coordinates of the chain in the ChainShape itself, we are choosing the ease of redundancy and keeping our own list of those points as well. Later, when we go to draw the ChainShape, we don't have to ask Box2D for the locations of the vertices.

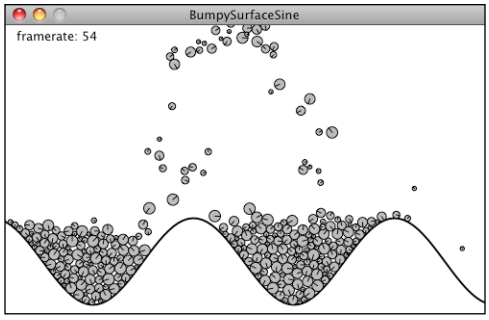
```
  void display() {
    strokeWeight(1);
    stroke(0);
    noFill();
    beginShape();
    for (Vec2 v: surface) {
      vertex(v.x,v.y);              $$ Draw the ChainShape as a series of vertices
    }
    endShape();
  }
}
```

What we need in the main tab for our Surface object is quite simple, given that Box2D takes care of all of the physics for us.

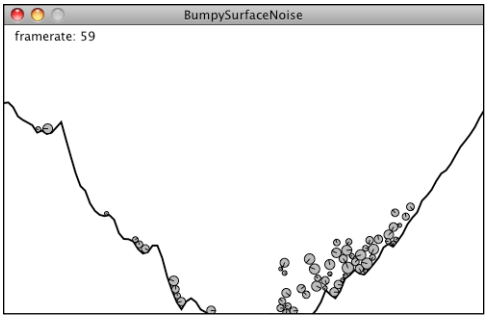
```
PBox2D box2d;

Surface surface;

void setup() {
  size(500,300);
  box2d = new PBox2D(this);
  box2d.createWorld();
```

```
  surface = new Surface();          $$ Make a Surface object
}

void draw() {
  box2d.step();

  background(255);
  surface.display();                $$ Draw the Surface
}
```

Exercise: Review how we learned to draw a wave pattern in Chapter 3. Create an ChainShape out of a sine wave. Try using Perlin noise (see Prologue) as well.



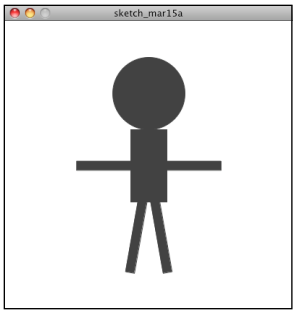
Sine Wave



Perlin Noise

5.10 Complex Forms

Now that we've seen how easy it is to make simple geometric forms in Box2D, let's imagine that you want to have a more complex form, such as a little alien stick figure.



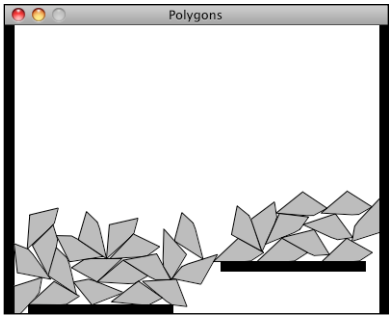
There are two strategies in Box2D for making forms that are more advanced than a basic circle or square. One is to use a PolygonShape in a different way. In our previous examples, we used PolygonShape to generate a rectangular shape with the ***setAsBox()*** function.

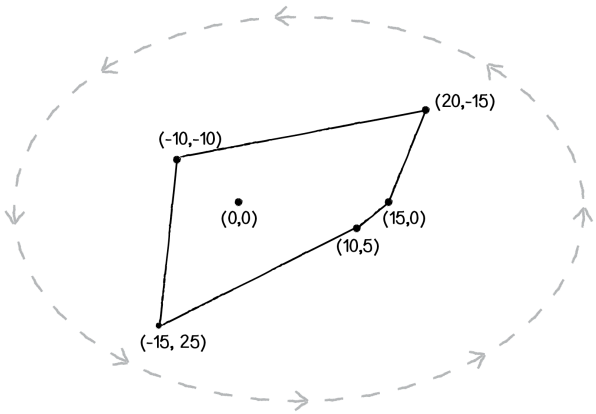
```
PolygonShape ps = new PolygonShape();
ps.setAsBox(box2dW, box2dH);
```

This was a good way to start because of the inherent simplicity of working with rectangles. However, PolygonShape can also be generated from a array of vectors**,** which allows you to build a completely custom shape as a series of connected vertices. This works very similarly to ChainShape.

Example: Polygon Shapes

```
Vec2[] vertices = new Vec2[4];      $$ An array of 4 vectors
vertices[0] = box2d.vectorPixelsToWorld(new Vec2(-15, 25));
vertices[1] = box2d.vectorPixelsToWorld(new Vec2(15, 0));
vertices[2] = box2d.vectorPixelsToWorld(new Vec2(20, -15));
vertices[3] = box2d.vectorPixelsToWorld(new Vec2(-10, -10));

PolygonShape ps = new PolygonShape();
ps.set(vertices, vertices.length);
                  $$ Making a polygon from that array
```

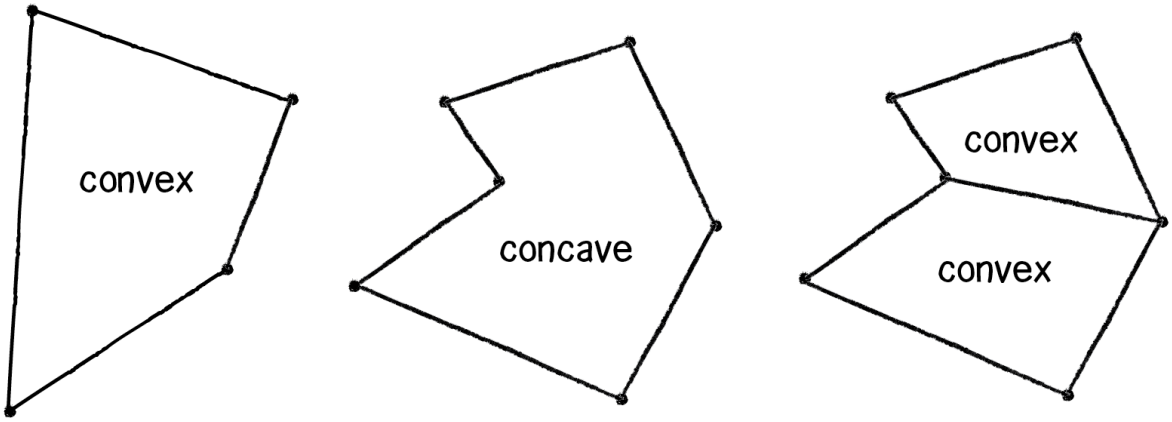




When building your own polygon in Box2D, you must remember two important details.

1. Order of vertices! If you are thinking in terms of pixels (as above) the vertices should be defined in counter-clockwise order. (When they are translated to Box2D world vectors, they will actually be in clockwise order since the vertical axis is flipped.)

2. Convex shapes only! A concave shape is one where the surface curves inward. Convex is the opposite (see illustration below). Note how in a concave shape every internal angle must be 180 degrees or less. Box2D is not capable of handling collisions for "concave" shapes. If you need a concave shape, you will have to build one out of multiple convex shapes (more about that in a moment).



a concave shape can be drawn with multiple convex shapes

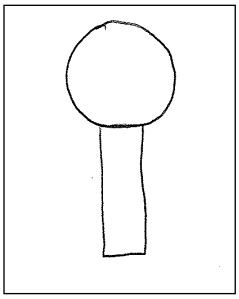
Now, when it comes time to display the shape in Processing, we can no longer just use ***rect()*** or ***ellipse()***. Since the shape is built out of custom vertices, we'll want to use Processing's ***beginShape()***, ***endShape()***, and ***vertex()*** functions. As we saw with the ChainShape, we could choose to store the pixel locations of the vertices in our own ArrayList for drawing. However, it's also useful to see how we can ask Box2D to report back to use the vertex locations.

```
void display() {
  Vec2 pos = box2d.getBodyPixelCoord(body);
  float a = body.getAngle();

  Fixture f = body.getFixtureList();        $$ First we get the Fixture attached to the Body
  PolygonShape ps = (PolygonShape) f.getShape();   $$ Then the Shape attached to the Fixture

  rectMode(CENTER);
  pushMatrix();
  translate(pos.x,pos.y);
  rotate(-a);
  fill(175);
  stroke(0);
  beginShape();
  for (int i = 0; i < ps.getVertexCount(); i++) {       $$ We can loop through that array and
    Vec2 v = box2d.vectorWorldToPixels(ps.getVertex(i));    convert each vertex from Box2D
    vertex(v.x,v.y);                                        space to pixels.
  }
  endShape(CLOSE);
  popMatrix();
}
```

Exercise: Using PolygonShape, create your own Polygon design (remember, it must be concave). Some possibilities below.

[ILLUSTRATE SOME OTHER POLYGON EXAMPLES]

A polygon shape will get us pretty far in Box2D. Nevertheless, the convex shape requirement will severely limit the range of possibilities. The good news is that we can completely eliminate this limit by creating a single Box2D body out of multiple shapes! Let's return to our little alien creature and simplify the shape to be a thin rectangle with a circle on top.



How can we build a single Body with two Shapes? Let's first review how we built a single Body with one Shape.

Step 1. Define the Body
Step 2. Create the Body
Step 3. Define the Shape
Step 4. Attach the Shape to the Body
Step 5. Finalize the Body's mass

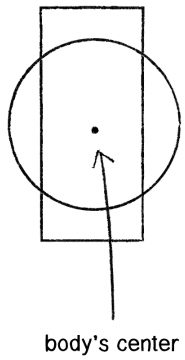
Attaching more than one Shape to a Body is as simple as repeating steps 3 and 4 over and over again.

Step 3a. Define Shape 1
Step 4a. Attach Shape 1 to the Body
Step 3b. Define Shape 2
Step 4b. Attach Shape 2 to the Body
etc. etc. etc.

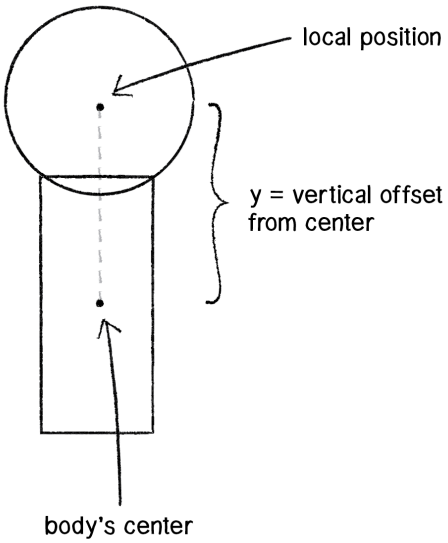
Let's see what this would look like with actual Box2D code.

```
BodyDef bd = new BodyDef();                          $$ Making the Body
bd.type = BodyType.DYNAMIC;
bd.position.set(box2d.coordPixelsToWorld(center));
body = box2d.createBody(bd);

PolygonShape ps = new PolygonShape();                $$ Making Shape 1 (the rectangle)
float box2dW = box2d.scalarPixelsToWorld(w/2);
float box2dH = box2d.scalarPixelsToWorld(h/2);
sd.setAsBox(box2dW, box2dH);

CircleShape cs = new CircleShape();                  $$ Making Shape 2 (the circle)
cs.m_radius = box2d.scalarPixelsToWorld(r);

body.createFixture(ps,1.0);                          $$ Attach both shapes with a Fixture
body.createFixture(cs, 1.0);
```

The above looks pretty good, but sadly, if we run it, we'll get the following result:



When you attach a Shape to a Body, by default, the center of the Shape will be located at the center of the Body. But in our case, if we take the center of the rectangle to be the center of the Body, we want the center of the circle to be offset along the y-axis from the Body's center.



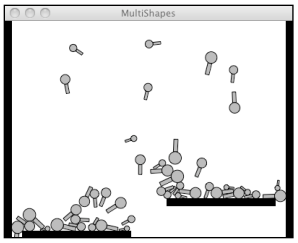
This is achieved by using the local position of a Shape, accessed via a Vec2 variable called ***m_p***.

```
Vec2 offset = new Vec2(0,-h/2);      $$ Our offset in pixels
offset = box2d.vectorPixelsToWorld(offset);      $$ Converting the vector to Box2D world
circle.m_p.set(offset.x,offset.y);               $$ Setting the local position of the circle
```

Then when we go to draw the Body, we use both ***rect()*** and ***ellipse()*** with the circle offset the same way.

Example: Multiple Shapes on one Body

```
void display() {
  Vec2 pos = box2d.getBodyPixelCoord(body);
  float a = body.getAngle();

  rectMode(CENTER);
  pushMatrix();
  translate(pos.x,pos.y);
  rotate(-a);
  fill(175);
  stroke(0);
  rect(0,0,w,h);           $$ First the rectangle at (0,0)
  ellipse(0,-h/2,r*2,r*2); $$ Then the ellipse offset at (0,-h/2)
  popMatrix();
}
```

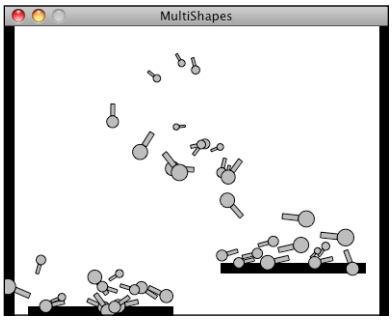


Finishing off this section, I want to stress the following: the stuff you draw in your Processing window doesn't magically experience physics simply because we created some Box2D Bodies and Shapes. These examples work because we very carefully matched how we draw our elements with how we defined the Bodies and Shapes we put into the Box2D world. If you accidentally draw your shape differently,you won't get an error, not from Processing or from Box2D. However, your sketch will look odd and the physics won't work correctly. For example, what if we had written:

```
Vec2 offset = new Vec2(0,-h/2);
```

when we created the Shape, but:

```
ellipse(0,h/2,r*2,r*2);
```

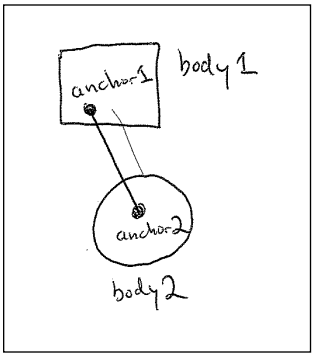


when it came time to display the Shape? The results would look like the image to the right, where clearly, the collisions are not functioning as expected. This is not because the physics is broken; it's because we did not communicate properly with Box2D either when we put stuff in the magic world or queried the world for locations.

Exercise: Make your own little alien being using multiple Shapes attached to a single Body. Try using more than one Polygon to make a concave shape. Remember, you aren't limited to using the shape drawing functions in Processing; you can use images, colors, add hair with lines, etc. Think of the Box2D shapes as only a skeleton for your creative and fantastical design!

5.11 Feeling Attached—Box2D Joints

Box2D joints allow you to connect one Body to another, enabling more advanced simulations of swinging pendulums, elastic bridges, squishy characters, wheels spinning on an axle, etc. There are many different kinds of Box2D joints. In this chapter we're going to look at three: distance joints, revolute joints, and "mouse" joints.



Let's begin with a distance joint, a joint that connects two Bodies with a fixed length. The joint is attached to each Body at a specified anchor point (a point relative to the Body's center.) For any Box2D joint, we need to follow these steps. This, of course, is similar to the methodology we used to build Bodies and Shapes, with some quirks.

Step 1. Make sure you have two Bodies ready to go.
Step 2. Define the Joint.
Step 3. Configure the Joint's properties (What are the Bodies? Where are the anchors? What is its rest length? Is it elastic or rigid?)
Step 4. Create the Joint.

Let's assume we have two Particle objects that each store a reference to a Box2D Body. We'll call them Particles p1 and p2.

```
Particle p1 = new Particle();
Particle p2 = new Particle();
```

OK, onto Step 2. Let's define the Joint.

```
DistanceJointDef djd = new DistanceJointDef();
```

Easy, right? Now it's time to configure the Joint. First we tell the Joint which two Bodies it connects:

```
djd.bodyA = p1.body;
djd.bodyB = p2.body;
```

Then we set up a rest length. Remember, if our rest length is in pixels, we need to convert it!

```
djd.length = box2d.scalarPixelsToWorld(10);
```

A distance joint also includes two optional settings that can make the joint soft, like a spring connection: frequencyHz and damping ratio.

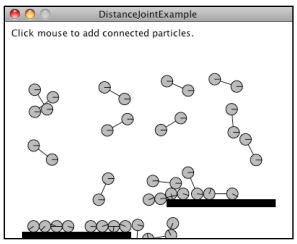
```
djd.frequencyHz  = ___;  $$ Measured in Hz, like the frequency of harmonic oscillation; try
                            values between 1 and 5
djd.dampingRatio = ___;  $$ Dampens the spring, typically a number between 0 and 1
```

Finally, we create the Joint.

```
DistanceJoint dj = (DistanceJoint) box2d.world.createJoint(djd);
```

Box2D won't keep track of what kind of Joint we are making, so we have to cast it as a DistanceJoint upon creation.

We can create Box2D joints anywhere in our Processing sketch. Here's an example of how we might write a class to describe two Box2D bodies connected with a single joint.

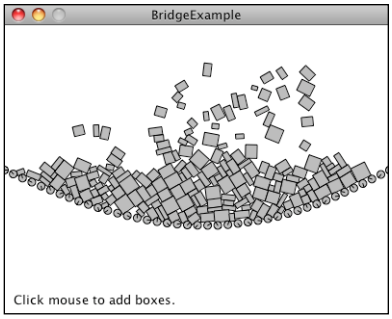


Example: DistanceJoint

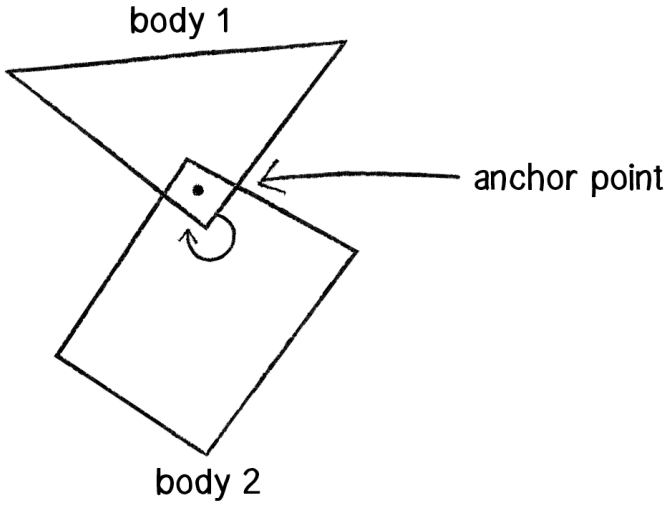
```
class Pair {

  Particle p1;          $$ Two objects that each have a Box2D body
  Particle p2;
  float len = 32;       $$ Arbitrary rest length

  Pair(float x, float y) {

    p1 = new Particle(x,y);
    p2 = new Particle(x+random(-1,1),y+random(-1,1));  $$ Problems can result if the bodies are
                                                          initialized at the same location

    DistanceJointDef djd = new DistanceJointDef();     $$ Making the joint!
    djd.bodyA = p1.body;
    djd.bodyB = p2.body;
    djd.length = box2d.scalarPixelsToWorld(len);
    djd.frequencyHz = 0;  // Try a value less than 5
    djd.dampingRatio = 0; // Ranges between 0 and 1

    DistanceJoint dj = (DistanceJoint) box2d.world.createJoint(djd);
  }                              $$ Make the joint.  Note we aren't storing a reference to the
                                 joint anywhere! We might need to someday, but for now it's OK.

  void display() {
    Vec2 pos1 = box2d.getBodyPixelCoord(p1.body);
    Vec2 pos2 = box2d.getBodyPixelCoord(p2.body);
    stroke(0);
    line(pos1.x,pos1.y,pos2.x,pos2.y);
```

```
    p1.display();
    p2.display();
  }
}
```

Exercise: Create a simulation of a bridge by using distance joints to connect a sequence of circles (or rectangles) as illustrated to the right. Assign a density of zero to lock the endpoints in place. Experiment with different values to make the bridge more or less "springy." It should also be noted that the joints themselves have no physical geometry, so in order for your bridge not to have holes, spacing between the nodes will be important.



Another joint you can create in Box2D is a *Revolute Joint*.



A revolute connects two Box2D bodies at a common anchor point, which can also be referred to as a "hinge." The joint has an "angle" which describes the relative rotation of each Body. To use a Revolute Joint, we follow the same steps we did with the Distance Joint.

Step 1. Make sure you have two bodies ready to go.

Let's assume we have two "Box" objects, each of which stores a reference to a Box2D Body.

```
Box box1 = new Box();
Box box2 = new Box();
```

Step 2. Define the Joint.

Now we want a RevoluteJointDef.

```
RevoluteJointDef rjd = new RevoluteJointDef();
```

Step 3. Configure the Joint's properties.

The most important properties of a RevoluteJoint are the two bodies it connects as well as their mutual anchor point (i.e where they are connected). There are set with the function ***initialize()***.

```
rjd.initialize(box1.body, box2.body, box1.body.getWorldCenter());
```

Notice how the first two arguments specify the bodies and the second point specifies the anchor, which in this case is located at the center of the first body.

An exciting aspect to the RevoluteJoint is that you can motorize it so it spins autonomously. For example:

```
rjd.enableMotor = true;      $$ Turn on the motor.
rjd.motorSpeed = PI*2;       $$ How fast is the motor?
rjd.maxMotorTorque = 1000.0; $$ How powerful is the motor?
```

The motor can be enabled and disabled while the program is running.

Finally, the ability for a RevoluteJoint to spin can be constrained between two angles (By default, it can rotate a full 360 degrees, or TWO_PI radians.)

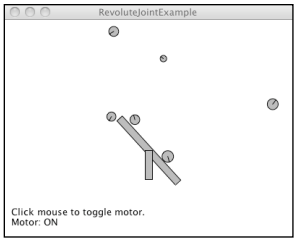
```
rjd.enableLimit = true;
rjd.lowerAngle = -PI/8;
rjd.upperAngle = PI/8;
```

Step 4. Create the Joint.

```
RevoluteJoint joint = (RevoluteJoint) box2d.world.createJoint(rjd);
```

Let's take a look at all of these steps together in a class called ***Windmill***, which connects two boxes with a revolute joint. In this case, "box1" has a density of zero, so only "box2" spins around a fixed point.

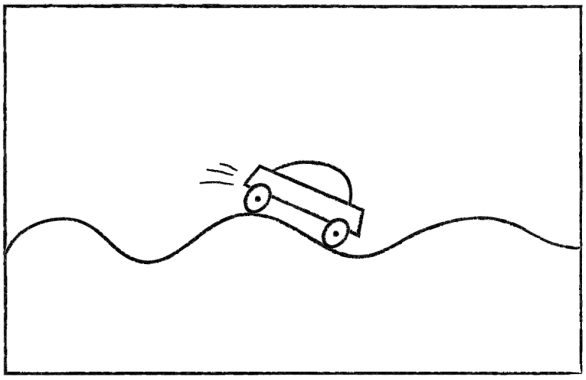
Example: Spinning Windmill




```
class Windmill {

  RevoluteJoint joint;  $$ Our "Windmill" is two boxes and one joint
  Box box1;
  Box box2;

  Windmill(float x, float y) {

    box1 = new Box(x,y,120,10,false);
```

```
    box2 = new Box(x,y,10,40,true);   $$ In this example, the Box class expects a boolean
                                      argument argument that will be used to determine if the
                                      Box is fixed or not.  See web site for the Box class code.

    RevoluteJointDef rjd = new RevoluteJointDef();
    rjd.initialize(box1.body, box2.body, box1.body.getWorldCenter());
                                      $$ The Joint connects two Bodies and is anchored at the
                                         center of the first body.

    rjd.motorSpeed = PI*2;       $$ A Motor!
    rjd.maxMotorTorque = 1000.0;
    rjd.enableMotor = true;

    joint = (RevoluteJoint) box2d.world.createJoint(rjd);  $$ Create the Joint
  }

  void toggleMotor() {       $$ Turning the motor on or off
    boolean motorstatus = joint.isMotorEnabled();
    joint.enableMotor(!motorstatus);
  }

  void display() {
    box1.display();
    box2.display();
  }
}
```

Exercise: Use a RevoluteJoint for the wheels of a Car. Use motors so that the car drives autonomously. Try using an ChainShape for the road's surface.



The last joint we'll look at is the MouseJoint. The MouseJoint is typically used for moving a Body with the mouse. However, it can also be used to drag an object around the screen according to some arbitrary x and y. The joint functions by pulling the Body towards a "target" position.

Before we look at the MouseJoint itself, let's ask ourselves why we even need it in the first place. If you look at the Box2D documentation, there is a function called ***setTransform()*** that specifically "sets the position of the body's origin and rotation (radians)." If a Body has a position, can't we just assign the Body's position to the mouse?

```
Vec2 mouse = box2d.screenToWorld(x,y);
```

```
body.setTransform(mouse,0);
```

While this will in fact move the Body, it also will have the unfortunate result of breaking the physics. Let's imagine you built a teleportation machine that allows you to teleport from your bedroom to your kitchen (good for late night snacking.) Now, go ahead and rewrite Newton's laws of motion to account for the possibility of teleportation. Not so easy, right? Box2D has the same problem. If you manually assign the location of an body, it's like saying "teleport that body" and Box2D no longer knows how to compute the physics properly. However, Box2D does allow you to tie a rope to yourself and get a friend of yours to stand in the kitchen and drag you there. This is what the Mouse Joint does. It's like a string you attach to a Body and pull towards a target.

Let's look at making this joint, assuming we have a Box object: *box*. This code will look identical to our distance joint with one small difference.

```
MouseJointDef md = new MouseJointDef();        $$ Just like before, define the Joint

md.bodyA = box2d.getGroundBody();              $$ Whoa, this is new!
md.bodyB = box.body;                           $$ Attach the Box Body

md.maxForce = 5000.0;                          $$ Set properties
md.frequencyHz = 5.0;
md.dampingRatio = 0.9;

MouseJoint mouseJoint = (MouseJoint) box2d.world.createJoint(md);     $$ Create the Joint
```

So, what's this line of code all about?

```
md.bodyA = box2d.getGroundBody();
```

Well, as we've stated, a joint is a connection between *two* bodies. With the MouseJoint, we're saying that the second body is, well, the ground. Hmm. What the heck is the *ground* in Box2D? One way to imagine it is to think of the screen as the ground. What we're doing is making a joint that connects a rectangle drawn on the window with the Processing window itself. And the point in the window to which the connection is tied is a moving target.

Once we have a MouseJoint, we'll want to update the target location continually while the sketch is running.

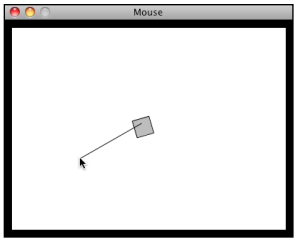
```
Vec2 mouseWorld = box2d.coordPixelsToWorld(mouseX,mouseY);
mouseJoint.setTarget(mouseWorld);
```

To make this work in an actual Processing sketch, we'll want to have the following:

- ***Box class***—*An object that references a Box2D Body.*
- ***Spring class***—*An object that manages the MouseJoint that drags the Box object around.*

- ***Main tab***—*Whenever mousePressed() is called, the MouseJoint is created; whenever mouseReleased() is called, the MouseJoint is destroyed. This allows us to interact with a Body only when the mouse is pressed.*

Let's take a look at the main tab. You can find the rest of the code for the Box and Spring classes on the book web site.



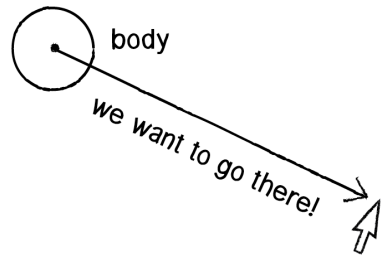

```
Example: MouseJoint demonstration
PBox2D box2d;

Box box;          $$ One Box
Spring spring;    $$ Object to manage MouseJoint

void setup() {
  size(400,300);
  box2d = new PBox2D(this);
  box2d.createWorld();

  box = new Box(width/2,height/2);
  spring = new Spring();      $$ The MouseJoint is really null until we click the mouse
}

void mousePressed() {
  if (box.contains(mouseX, mouseY)) {    $$ Was the mouse clicked inside the Box?
    spring.bind(mouseX,mouseY,box);      $$ If so, attach the MouseJoint
  }
}

void mouseReleased() {
  spring.destroy();          $$ When the mouse is released, we're done with the Joint
}

void draw() {
  background(255);

  box2d.step();

  spring.update(mouseX,mouseY);    $$ We must always update the MouseJoint's target

  box.display();
  spring.display();
}
```

Exercise: Use a MouseJoint to move a Box2D Body around the screen according to an algorithm or input other than the mouse. For example, assign it a location according to Perlin noise or key presses. Or build your own controller using an Arduino (http://www.arduino.cc/).

It's worth noting that while the technique for dragging an object around using a MouseJoint is useful, Box2D also allows a Body to have a "kinematic" body type.

```
BodyDef bd = new BodyDef();
bd.type = BodyType.KINEMATIC;      $$ Setting the body type to Kinematic
```

Kinematic bodies can be controlled by the user through setting their velocity directly. For example, let's say you want an object to follow a target (like your mouse). You could create a vector that points from a body's location to a target.




```
Vec2 pos = body.getWorldCenter();
Vec2 target = box2d.coordPixelsToWorld(mouseX,mouseY);
Vec2 v = target.sub(pos);     $$ A vector pointing from the Body position to the Mouse
```

Once you have that vector, you could assign it to the body's velocity so that it moves to the target.

```
body.setLinearVelocity(v);    $$ Assigning a body's velocity directly, overriding physics!
```

You can also do the same with angular velocity (or leave it alone and allow the physics to take over).

It is important to note that kinematic bodies do not collide with other kinematic or static bodies. In these cases, the MouseJoint strategy is preferable.

Exercise: Redo your previous exercise, but use a Kinematic body instead.

5.12 Bringing it all back home to forces

In Chapter 2, we spent a lot of time thinking about building environments with multiple forces. An object might respond to gravitational attraction, wind, air resistance, etc. Clearly there are forces at work in Box2D as we watch rectangles and circles spin and fly around the screen. But so far, we've only had the ability to manipulate a single global force—gravity.

```
box2d = new PBox2D(this);
box2d.createWorld();
box2d.setGravity(0, -20);        $$ Setting the global gravity force
```

If we want to use any of our Chapter 2 techniques with Box2D, we need look no further than our trusty ***applyForce()*** function. In our Mover class we wrote a function called ***applyForce(),*** which received a vector, divided it by mass, and accumulated it into the Mover's acceleration.

With Box2D, the same function exists, but we don't need to write it ourselves. Instead, we can call the Box2D Body's ***applyForce()*** function!

```
class Box {
  Body body;

  // etc. etc.
  void applyForce(Vec2 force) {
    Vec2 pos = body.getWorldCenter();
    body.applyForce(force, pos);
  }
}
```

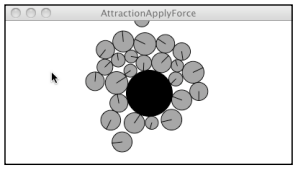
Here we are receiving a force vector and passing it along to the Box2D Body object. The key difference is that Box2D is a more sophisticated engine that our examples from Chapter 2. Our earlier forces examples assumed that the force was always applied at the Mover's center. Here we get to specify exactly where on the Body the force is applied. In the above code, we're just applying it to the center by asking the Body for its center, but this could be adjusted.

Let's say we wanted to use a gravitational attraction force. Remember the code we wrote back in Chapter 2 in our Attractor class?

```
  PVector attract(Mover m) {
    PVector force = PVector.sub(location,m.location);
    float distance = force.mag();
    distance = constrain(distance,5.0,25.0);
    force.normalize();
    float strength = (g * mass * m.mass) / (distance * distance);
    force.mult(strength);
    return force;
  }
```

We can rewrite the exact same function using Vec2 instead and use it in a Box2D example. Note how for our force calculation we can stay completely within the Box2D coordinate system and never think about pixels.

```
  Vec2 attract(Mover m) {
    Vec2 pos = body.getWorldCenter();          $$ We have to ask Box2D for the locations first!
    Vec2 moverPos = m.body.getWorldCenter();
    Vec2 force = pos.sub(moverPos);

    float distance = force.length();
    distance = constrain(distance,1,5);
    force.normalize();
    float strength = (G * 1 * m.body.m_mass) / (distance * distance);
    force.mulLocal(strength);      $$ Remember, it's mulLocal() for Vec2
    return force;
  }
```



Exercise: Take any example you made previously using a force calculation and bring that force calculation into Box2D.

5.13 Collision Events

Now we've seen a survey of what can be done with Box2D. Since this book is not called "The Nature of Box2D", it's not my intention to cover every single possible feature of the Box2D engine. But hopefully by looking at the basics of building bodies, shapes, and joints, when it comes time to use an aspect of Box2D that we haven't covered, the skills we've gained here will make that process considerably less painful. There is one more feature of Box2D, however, that I do think is worth covering.

Let's ask a question you've likely been wondering:

What if I want something to happen when two Box2D bodies collide? I mean, don't get me wrong—I'm thrilled that Box2D is handling all of the collisions for me. But if it takes care of everything for me, how am I supposed to know when things are happening?

Your first thoughts when considering an event during which two objects collide might be as follows: Well, if I know all the bodies in the system, and I know where they are all located, then I can just start comparing the locations, see which ones are intersecting, and determine that they've collided. That's a nice thought, but hello??!? The whole point of using Box2D is that Box2D will take care of that for us. If we are going to do the geometry to test for intersection ourselves, then all we're doing is re-implementing Box2D.

Of course, Box2D has thought of this problem before. It's a pretty common one. After all, if you intend to make a bajillion dollars selling some game called Angry Birds, you better well make something happen when an ill-tempered pigeon smashes into a cardboard box. Box2D alerts you to moments of collision with something called an "interface." It's worth learning about interfaces, an advanced feature of object-oriented programming. You can take a look at the Java Interface tutorial (http://download.oracle.com/javase/tutorial/java/concepts/interface.html) as well as the JBox2D ContactListener class. (I have also included an example on the web site that demonstrates using the interface directly.)

If you are using PBox2D, as we are here, you don't need to implement your own interface. Detecting collision events is done through a callback function no different than mousePressed().

```
void mousePressed() {                    $$ The mousePressed event with which we are comfortable
  println("The mouse was pressed!");
}


void beginContact(Contact cp) {          $$ What our "beginContact" event looks like
  println("Something collided in the Box2D World!");
}
```

Before the above will work, you must first let PBox2D know you intend to listen for collisions. (This allows the library to reduce overhead by default; it won't bother listening if it doesn't have to.)

```
void setup() {
  box2d = new PBox2D(this);
  box2d.createWorld();
  box2d.listenForCollisions();     $$ Add this line if you want to listen for collisions
}
```

There are four collision event callbacks.

- ***beginContact()***—this is triggered whenever two shapes first come into contact with each other.
- ***endContact()—***this is triggered over and over again as long as shapes continue to be in contact.
- ***preSolve()—***this is triggered before Box2D solves the outcome of the collision (i.e. before beginContact()). It can be used to disable a collision if necessary.
- ***postContact()—***this is triggered after the outcome of the collision is solved and allows you to gather information about that "solution" (known as an "impulse").

The details behind ***preSolve()*** and ***postSolve()*** are beyond the scope of this book, however, we are going to take a close look at ***beginContact()*** which will cover the majority of conventional cases in which you want to trigger an action when a collision occurs. ***endContact()*** works identically to ***beginContact()***, the only difference being that it occurs the moment bodies separate.

Ok, ***beginContact()*** is written as follows:

```
void beginContact(Contact cp) {

}
```

Notice that the function above includes an argument of type ***Contact***. A Contact object includes all the data associated with a collision—the geometry and the forces. Let's say we have a Processing sketch with Particle objects that store a reference to a Box2D body. Here is the process we are going to follow.

1. Contact, could you tell me what two things collided?

Now, what has collided here? Is it the bodies? The shapes? The fixtures? Box2D detects collisons between shapes, after all, these are the entities that have geometry. However, because shapes are attached to bodies with fixtures, what we really want to ask Box2D is: "Could you tell me which two fixtures collided?"

```
Fixture f1 = cp.getFixtureA();     $$ The Contact stores the fixtures as A and B
Fixture f2 = cp.getFixtureB();
```

2. Fixtures, could you tell me which Body you are attached to?

```
Body b1 = f1.getBody(); $$ getBody() gives us the Body that the Fixture is attached
Body b2 = f2.getBody();
```

3. Bodies, could you tell me which Particles you are associated with?

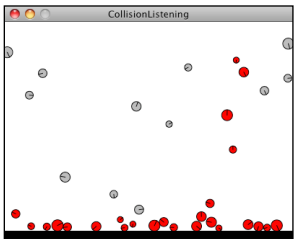
OK, this is the harder part. After all, Box2D doesn't know anything about our code. Sure, it is doing all sorts of stuff to keep track of the relationships between Shapes and Bodies and Joints, but it's up to us to manage our own objects and their associations with Box2D elements. Luckily for us, Box2D provides a function that allows us to attach our Processing object (a Particle) to a Box2D Body via the ***setUserData()*** and ***getUserData()*** methods.

Let's take a look at the constructor in our Particle class where the Body is made. We are expanding our Body-making procedure by one line of code, noted below.

```
class Particle {
  Body body;

  Particle(float x, float y, float r) {
    BodyDef bd = new BodyDef();
    bd.position = box2d.coordPixelsToWorld(x, y);
    bd.type = BodyType.DYNAMIC;
    body = box2d.createBody(bd);
    CircleShape cs = new CircleShape();
    cs.m_radius = box2d.scalarPixelsToWorld(r);
    body.createFixture(fd,1);

    body.setUserData(this);     $$ "this" refers to this Particle object.
                                We are telling the Box2D Body to store a
                                reference to this Particle that we can
                                access later.

  }
```

Later, in our ***addContact()*** function, once we know the Body, we can access the Particle object with ***getUserData().***

Example: CollisionListening




```
void beginContact(Contact cp) {

  Fixture f1 = cp.getFixtureA();
  Fixture f2 = cp.getFixtureB();

  Body b1 = f1.getBody();
  Body b2 = f2.getBody();

  Particle p1 = (Particle) b1.getUserData();
  Particle p2 = (Particle) b2.getUserData();   $$ When we pull the "user data" object out of the
                                               Body object, we have to remind our program that it
```

```
                                        is a Particle object.  Box2D doesn't know this.
  p1.change();
  p2.change();              $$ Once we have the particles, we can do anything to them.  Here we
                            just call a function that changes their color.
}
```

Now, in many cases, we cannot assume that the objects that collided are all Particle objects. We might have a sketch with Boundary objects, Particle objects, Box objects, etc. So often we will have to query the "user data" and find out what kind of object it is before proceeding.

```
  Object o1 = b1.getUserData();              $$ Getting a generic object
  if (o1.getClass() == Particle.class) {     $$ Asking that object if it's a Particle
    Particle p = (Particle) o1;
    p.change();
  }
```

Exercise: Consider how polymorphism could help in this case. Build an example in which several classes extend one class and therefore eliminate the need for the above testing.

It should also be noted that due to how Box2D triggers these callbacks, you cannot create or destroy Box2D entities inside of ***beginContact(), endContact(), preSolve(), or postSolve()***. If you want to do this, you'll need to set a variable inside an object (something like: *markForDeletion = true*), which you check during ***draw()*** and then delete objects.

Exercise: Create a simulation in which Particle objects disappear when they collide. Use the methodology I just described.

5.14 A Brief Interlude -- Integration Methods

Has the following ever happened to you? You're at a fancy cocktail party regaling your friends with tall tales of software physics simulations. Someone pipes up: "Enchanting! But what integration method are you using?" "What?!" you think to yourself. "Integration?"

Maybe you've heard the term before. Along with "differentiation," it's one of the two main operations in calculus. Right, calculus. The good news is, we've gotten through about 90% of the material in this book related to physics simulation and we haven't really needed to dive into calculus. But as we're coming close to finishing this topic, it's worth taking a moment to examine the calculus behind what we have been doing and how it relates to the methodology in certain physics libraries (like Box2D and the upcoming toxiclibs).

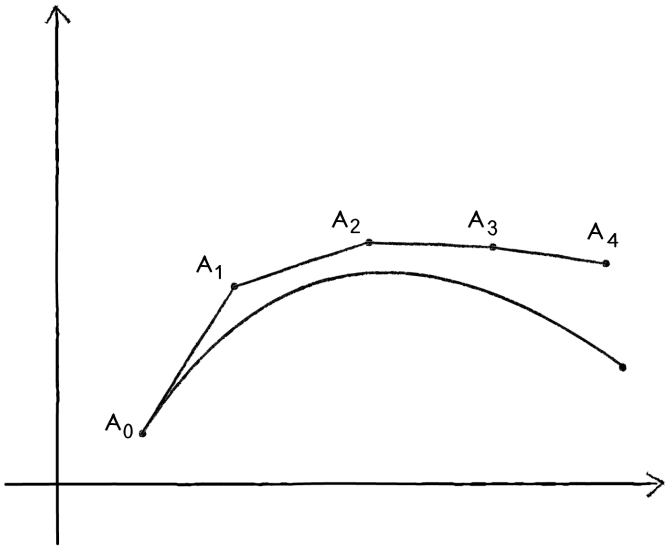
Let's begin by answering the question: "What does integration have to do with location, velocity, and acceleration?" Well, first let's define differentiation. The derivative of a function is a measure of how a function changes over time. Consider location and its derivative. Location is a point in space, while velocity is change in location over time. Therefore, velocity can be described as the "derivative" of location. What is acceleration? The change in velocity over time—i.e. the "derivative" of velocity.

Now that we understand the derivative (differentiation), we can define the integral (integration) as the inverse of the derivative. In other words, the integral of an object's velocity over time tells us the object's new location when that time period ends. Location is the integral of velocity, and velocity is the integral of acceleration. Sinceour physics simulation is founded upon the process of calculating acceleration based on forces , we need integration to figure out where the object is after a certain period of time (like one frame of animation!).

So we've been doing integration all along! It looks like this:

```
velocity.add(acceleration);
location.add(velocity);
```

The above methodology is known as Euler Integration (named for the mathematician Leonhard Euler, pronounced "Oiler") or the Euler Method. It's essentially the simplest form of integration and very easy to implement in our code (see the two lines above!). However, it is not necessarily the most efficient form, nor is it close to being the most accurate. Why is Euler inaccurate? Let's think about it this way. When you drive a car down the road pressing the gas pedal with your foot and accelerating, does the car sit in one location at time equals 1 second, then disappear and suddenly reappear in a new location at time equals 2 seconds, and do the same thing for 3 seconds, and 4, and 5? No, of course not. The car moves continuously down the road. But what's happening in our Processing sketch? A circle is at one location at frame 0, another at frame 1, another at frame 2. Sure, at 30 frames per second, we're seeing the illusion of motion. But we only calculate a new location every N units of time, whereas the real world is perfectly continuous. This results in some inaccuracies, as shown in the diagram below:



[DELETE A1, A2, etc. Label curve as "the real world",' the segements as "Euler Integration"]

The “real world” is the curve; Euler simulation is the series of line segments.

One option to improve on Euler is to use smaller timesteps—instead of once per frame, we could recalculate an object’s location twenty times per frame. But this isn’t practical; our sketch would then run too slowly.

I still believe that Euler is the best method for learning the basics, and it’s also perfectly adequate for most of the projects we might make in Processing. Anything we lose in efficiency or inaccuracy we make up in ease of use and understandability. For better accuracy, Box2D uses something called symplectic Euler (or semi-explicit Euler), a slight modification of Euler (see: http://en.wikipedia.org/wiki/Symplectic_Euler_method).

There is also an integration method called Runge–Kutta (named for German mathematicians C. Runge and M.W. Kutta), which is used in some physics engines.

A very popular integration method that our next physics library uses is known as “Verlet Integration.” A simple way to describe Verlet Integration is to think of our typical motion algorithm without velocity. After all, we don't really need to store the velocity. If we always know where an object was at one point in time and where it is now, we can extrapolate its velocity. Verlet Integration does precisely this, though instead of having a variable for velocity, it calculates velocity while the program is running. Verlet Integration is particularly well suited for particle systems, especially particle systems with spring connections between the particles. We don't need to worry about the details because toxiclibs, as we’ll see below, takes care of them for us. However, if you are interested, here is the seminal paper on Verlet physics, from which just about every Verlet computer graphics simulation is derived:

http://www.gamasutra.com/resource_guide/20030121/jacobson_pfv.htm

And of course, you can find out more about Verlet Integration via Wikipedia:

http://en.wikipedia.org/wiki/Verlet_integration

5.15 Verlet Physics with Toxiclibs

From toxiclibs.org:

“toxiclibs is an independent, open-source library collection for computational design tasks with Java & Processing developed by Karsten “toxi” Schmidt (thus far). The classes are purposefully kept fairly generic in order to maximize re-use in different contexts ranging from generative design, animation, interaction/interface design, data visualization to architecture and digital fabrication, use as teaching tool and more.”

In other words, we should thank our lucky stars for toxiclibs. We are only going to focus on a few examples related to Verlet physics, but toxiclibs includes a suite of other wonderful packages that help with audio, color, geometry, and more. In particular, if you are looking to work with form and fabrication in Processing, take a look at the geometry package. Many demos can be found here:

http://www.openprocessing.org/portal/?userID=4530

We should note that toxiclibs was designed specifically for use with Processing. This is great news. The trouble we had with making Box2D work in Processing (multiple coordinate systems, Box2D vs JBox2D vs PBox2D) is not an issue here. toxiclibs is a library that you just download, stick in your libraries folder, and use. And the coordinate system that we'll use for the physics engine is the coordinate system of Processing, so no translating back and forth. In addition, toxiclibs is not limited to a 2D world and all of the physics simulations and functions work in both two and three dimensions. So how do you decide which library you should use? Box2D or toxiclibs? If you fall into one of the following two categories, your decision is a bit easier:

My project involves collisions. I have circles, squares, and other strangely shaped objects that knock each other around and bounce off each other.

In this case, you are going to need Box2D. toxiclibs does not handle collisions.

My project involves lots of particles flying around the screen. Sometimes they attract each other. Sometimes they repel each other. And sometimes they are connected with springs.

In this case, toxiclibs is likely your best choice. It is simpler to use than Box2D and particularly well suited to connected systems of particles. Toxiclibs is also very high performance, due to the speed of the Verlet integration algorithm (not to mention the fact that the program gets to ignore all of the collision geometry.)

Here is a little chart that covers some of the features for each physics library.

Feature	Box2D	toxiclibs VerletPhysics
Collision geometry	Yes	No
3D physics	No	Yes
Particle attraction / repulsion forces	No	Yes
Spring connections	Yes	Yes
Other connections: revolute, pulley, gear, prismatic	Yes	No
Motors	Yes	No

Feature	Box2D	toxiclibs VerletPhysics
Friction	Yes	No

5.16 Getting toxiclibs

Everything you need to download and install toxiclibs can be found at:

http://toxiclibs.org/

When you download the library, you'll notice that it comes with eight modules (i.e. sub-folders), each a library in its own right. For the examples in this chapter, you will only need "verletphysics" and "toxiclibscore"; however, I recommend you take a look at and consider using all of the modules!

Once you have the library installed to your Processing library folder (see: http://wiki.processing.org/w/How_to_Install_a_Contributed_Library), you are ready to start looking at the following examples.

5.16 Core Elements of VerletPhysics

We spent a lot of time working through the core elements of a Box2D world: world, body, shape, joint. This gives us a head start on understanding toxiclibs, since it follows a similar structure.

Box2D	Toxiclibs VerletPhysics
World	VerletPhysics
Body	VerletParticle
Shape	Nothing! Toxiclibs does not handle shape geometry
Fixture	Nothing! Toxiclibs does not handle shape geometry
Joint	VerletSpring

5.17 Vectors with toxiclibs

Here we go again. Remember all that time we spent learning the ins and outs of ***PVector***? Then remember how when we got to Box2D, we had to translate all those concepts to a Box2D vector class: ***Vec2***? Well, it's time to do it again. toxiclibs also includes its own vector classes, one for two dimensions and one for three: ***Vec2D*** and ***Vec3D***.

Again, toxiclibs vectors are the same conceptually, but we need to learn a bit of new syntax. You can find all of the documentation for these vector classes here:

http://toxiclibs.org/docs/core/toxi/geom/Vec2D.html
http://toxiclibs.org/docs/core/toxi/geom/Vec3D.html

And let's just review some of the basic vector math operations with PVector translated to Vec2D (we're sticking with 2D for simplicity's sake).

PVector	Vec2D

`PVector a = new PVector(1,-1);`
`PVector b = new PVector(3,4);`
`a.add(b);`
	`Vec2D a = new Vec2D(1,-1);`
	`Vec2D b = new Vec2D(3,4);`
	`a.addSelf(b);`

`PVector a = new PVector(1,-1);`
`PVector b = new PVector(3,4);`
`PVector c = PVector.add(a,b);`
	`Vec2D a = new Vec2D(1,-1);`
	`Vec2D b = new Vec2D(3,4);`
	`Vec2D c = a.add(b);`

`PVector a = new PVector(1,-1);`
`float m = a.mag();`
`a.normalize();`
	`Vec2D a = new Vec2D(1,-1);`
	`float m = a.magnitude();`
	`a.normalize();`

5.17 Building the toxiclibs Physics World

The first thing we need to do to use VerletPhysics in our examples is import the library itself.

```
import toxi.physics2d.*;                    $$ Importing the libraries
import toxi.physics2d.behaviors.*;
import toxi.geom.*;
```

Then we'll need a reference to our physics world, a VerletPhysics or VerletPhysics2D object (depending on whether we are working in two or three dimensions.) The examples in this chapter will operate in 2D only for simplicity, but they could easily be extended into 3D (and 3D versions are available with the chapter download).

```
VerletPhysics2D physics;

void setup() {
  physics=new VerletPhysics2D();          $$ Creating a Toxiclibs verlet physics world
```

Once you have your Physics object, you can set some global properties for your world. For example, if you want it to have hard boundaries past which objects cannot travel, you can set its limits:

```
  physics.setWorldBounds(new Rect(0,0,width,height));
```

In addition, you can add gravity to the physics world with a GravityBehavior object. A GravityBehavior requires a vector—how strong and in what direction is the gravity?

```
  physics.addBehavior(new GravityBehavior(new Vec2D(0,0.5)));
}
```

Finally, in order to calculate the physics of the world and move the objects in the world, we have to call ***update()***. Typically this would happen once per frame in ***draw()***.

```
void draw() {
  physics.update();          $$ This is the same as Box2D's "step()" function
}
```

5.18 Adding Particles to the toxiclibs world

In the Box2D examples, we saw how we can create our own class (called, say, Particle) and include a reference to a Box2D Body.

```
class Particle {
  Body body;
```

This technique is somewhat redundant since Box2D itself keeps track of all of the Bodies in its world. However, it allows us to manage which Body is which (and therefore how each Body is drawn) without having to rely on iterating through Box2D's internal lists.

Let's look at how we might take the same approach with the class VerletParticle2D in toxiclibs. We want to make our own Particle class so that we can draw our Particle a certain way and include any custom properties. We'd probably write our code as follows:

```
class Particle {
  VerletParticle2D p;              $$ Our Particle has a reference to a VerletParticle

  Particle(Vec2D pos) {
    p = new VerletParticle2D(pos); $$ A VerletParticle needs an initial location (an x and y)
  }

  void display() {
    fill(0,150);
    stroke(0);
    ellipse(p.x,p.y,16,16);        $$ When it comes time to draw the Particle, we ask the
  }                                VerletParticle for its x and y coordinate
}
```

Looking at the above, we should first be thrilled to notice that drawing the Particle is as simple as grabbing the x and y and using them. No awkward conversions between coordinate systems here since toxiclibs is designed to think in pixels. Second, you might notice that this Particle class's sole purpose is to store a reference to a VerletParticle2D. This hints at something. Remember our discussion of inheritance back in Chapter 4: Particle Systems?

What is a Particle object other than an "augmented" VerletParticle? Why bother making a VerletParticle inside a Particle when we could simply extend VerletParticle?

```
class Particle extends VerletParticle2D {

  Particle(Vec2D loc) {
    super(loc);        $$ Calling super() so that the object is initialized properly
  }

  void display() {     $$ We want to be just like a VerletParticle, only with a display() method
    fill(175);
    stroke(0);
    ellipse(x,y,16,16); $$ We've inherited x and y from VerletParticle!
  }
}
```

Remember our multi-step process with the Box2D examples? We had to ask the Body for its location, then convert that location to pixels, then use that location in a drawing function. Now, because we have inherited everything from VerletParticle, our only step is to draw the shape at the x and y!

Incidentally, it's interesting to note that the VerletParticle2D class is a subclass of Vec2D. So in addition to inheriting everything from VerletParticle2D, our Particle class actually has all of the Vec2D functions available as well.

We can now create Particle objects anywhere within our sketch.

```
  Particle p1 = new Particle(new Vec2D(100,20));
  Particle p2 = new Particle(new Vec2D(100,180));
```

Just making a Particle object isn't enough, however. We have to make sure we tell our physics world about them with the ***addParticle()*** function.

```
  physics.addParticle(p1);
  physics.addParticle(p2);
```

If you look at the toxiclibs documentation, you'll see that the ***addParticle()*** expects a VerletParticle2D object.

addParticle(VerletParticle2D p)

And how can we then pass into the function our own "Particle" object? Remember that other tenet of object-oriented programming—polymorphism? Here, because our Particle class extends VerletParticle2D, we can choose to treat our Particle object in multiple ways—as a Particle or as a VerletParticle2D. This is an incredibly powerful feature of object-oriented programming. If we build our custom classes based on classes from toxiclibs, we can use our objects in conjunction with all of the functions toxiclibs has to offer.

5.19 Connecting Particles

toxiclibs has a set of classes that allow you to connect two VerletParticle objects with spring forces. There are three types of springs in toxiclibs:

- ***VerletSpring***: This class creates a springy connection between two VerletParticles in space. A Spring's properties can be configured in such a way as to create a stiff stick-like connection or a highly elastic stretchy connection. A Particle can also be locked so that only one end of the Spring can move.
- ***VerletConstrainedSpring***: A VerletConstrainedSpring is a spring whose maximum distance can be limited. This can help the whole spring system achieve better stability.
- ***VerletMinDistanceSpring***: A VerletMinDistanceSpring is a Spring that only enforces its rest length if the current distance is less than its rest length. This is handy if you want to ensure objects are at least a certain distance from each other, but don't care if the distance is bigger than the enforced minimum.

The inheritance and polymorphism technique we employed in the previous section proves also to be useful when creating VerletSprings. A VerletSpring expects two VerletParticles when the spring is created. And again, because our Particle class extends VerletParticle, VerletSpring will accept our Particles passed into the constructor. Let's take a look at some example code that assumes the existence of our two previous Particles p1 and p2 and creates a connection between them with a given rest length and strength.

```
float len = 80;              $$ What is the rest length of the spring?
float strength = 0.01;       $$ How strong is the spring?
VerletSpring2D spring=new VerletSpring2D(p1,p2,len,strength);
```

Just as with Particles, in order for the connection to actually be part of the physics world, we need to explicitly add it.

```
physics.addSpring(spring);
```

5.20 Putting it all together: A simple interactive Spring

One thing we saw with Box2D is that the physics simulation broke down when we overrode it and manually set the location of a Body. With toxiclibs' VerletPhysics, we don't have this problem. If we want to move the location of a Particle, we can simply set its x and y location manually. However, before we do so, it's generally a good idea to call the ***lock()*** function.

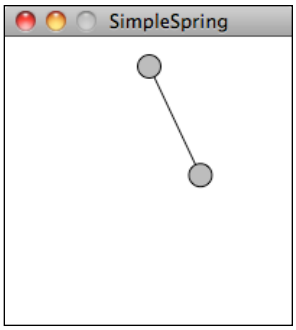
lock() is typically used to lock a Particle in place and is identical to setting a Box2D body's density to zero. However, here we are going to show how to lock a particle temporarily, move it, and then unlock it so that it continues to move according to the physics simulation.

Let's say you want to move a given particle whenever you click the mouse.

```
  if (mousePressed) {        $$ First lock the particle, then set the x and y, then unlock() it
    p2.lock();
    p2.x = mouseX;
    p2.y = mouseY;
    p2.unlock();
  }
```

And now we're ready to put all of these elements together in a simple example that connects two particles with a Spring. One Particle is locked in place, and the other can be moved by dragging the mouse. Note that this example is virtually identical to Example 3.x: Oscillating Spring (see p.XX ***[REF]***).

Example 15-x: Simple Spring with toxiclibs

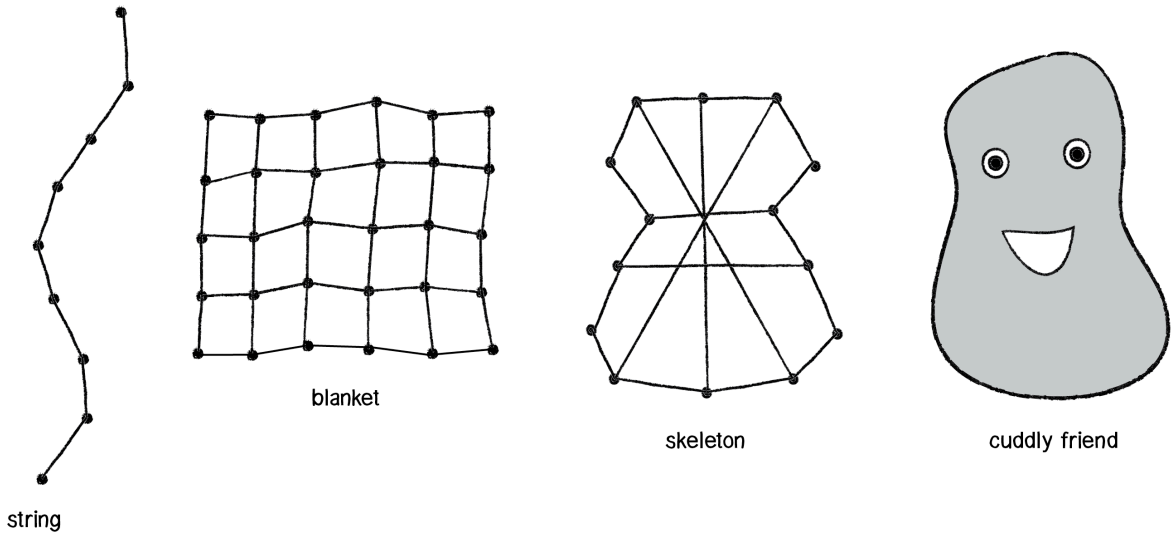



```
import toxi.physics2d.*;
import toxi.physics2d.behaviors.*;
import toxi.geom.*;

VerletPhysics2D physics;
Particle p1;
Particle p2;

void setup() {
  size(200,200);

  physics=new VerletPhysics2D();   $$ Creating a physics world
  physics.addBehavior(new GravityBehavior(new Vec2D(0,0.5)));
  physics.setWorldBounds(new Rect(0,0,width,height));

  p1 = new Particle(new Vec2D(100,20));  $$ Creating 2 Particles
  p2 = new Particle(new Vec2D(100,180));
  p1.lock();                             $$ Locking Particle 1 in place

  VerletSpring2D spring=new VerletSpring2D(p1,p2,80,0.01);$$ Creating 1 Spring

  physics.addParticle(p1);         $$ Must add everything to the world
  physics.addParticle(p2);
  physics.addSpring(spring);
}

void draw() {
  physics.update();                $$ Must update the physics

  background(255);

  line(p1.x,p1.y,p2.x,p2.y);       $$ Drawing everything
  p1.display();
  p2.display();

  if (mousePressed) {              $$ Moving a Particle according to the Mouse
    p2.lock();
    p2.x = mouseX;
    p2.y = mouseY;
    p2.unlock();
  }
}

class Particle extends VerletParticle2D {      $$ How cute is our simple Particle class!
```

```
  Particle(Vec2D loc) {
    super(loc);
  }

  void display() {
    fill(175);
    stroke(0);
    ellipse(x,y,16,16);
  }
}
```

5.20 Connected Systems Part I: String

The above example, two particles connected with a single spring, is the core building block for what toxiclibs' VerletPhysics is particularly well suited for: soft body simulations. For example, a string can be simulated by connecting a line of particles with springs. A blanket can be simulated by connecting a grid of particles with springs. And a cute, cuddly, squishy cartoon character can be simulated by a custom layout of particles connected with springs.

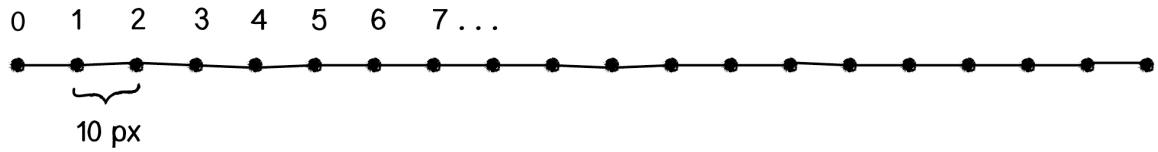


Let's begin by simulating a "soft pendulum"—a bob hanging from a string, instead of a rigid arm like we had in Chapter 3, Examples 3.x. Let's use Figure 3.x above as our model.

First, we'll need a list of Particle objects (let's use the same Particle class we built in the previous example).

```
ArrayList<Particle> particles = new ArrayList<Particle>();
```

Now, let's say we want to have 20 particles, all spaced 10 pixels apart.



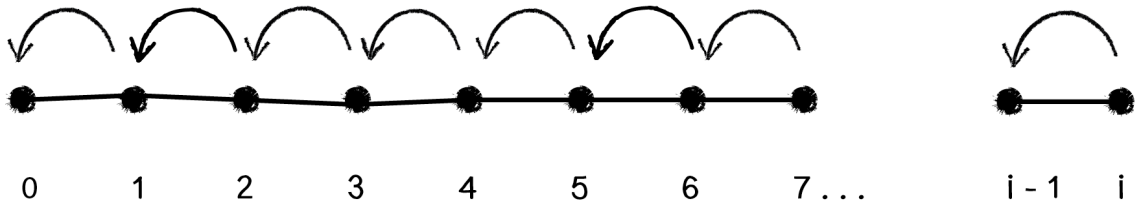

```
float len = 10;
float numParticles = 20;
```

We can loop from i equals 0 all the way up to 20, with each Particle's y location set to i * 10 so that the first particle is at (0,10), the second at (0,20), the third at (0,30), etc.

```
for(int i=0; i < numPoints; i++) {
  Particle particle=new Particle(i*len,10);    $$ Spacing them out along the x-axis
  physics.addParticle(particle);               $$ Add particle to our list
  particles.add(particle);                     $$ Add particle to physics world
}
```

Even though it's a bit redundant, we're going to add the Particle to both the toxiclibs physics world and to our own list. In case we eventually have multiple strings, this will allow us to know which particles are connected to which strings.

Now for the fun part: It's time to connect all the particles. Particle 1 will be connected to particle 0, particle 2 to particle 1, 3 to 2, 4 to 3, etc.



In other words, Particle i needs to be connected to Particle i-1 (except for when i = 0).

```
if (i != 0) {
 Particle previous = particles.get(i-1); $$ First we need a reference to the previous particle

 VerletSpring2D spring = new VerletSpring2D(particle,previous,len,strength);

                         $$ Then we make a spring connection between particle and previous
                         particle with a rest length and strength (both floats)

 physics.addSpring(spring);  $$ We must not forget to add the spring to the physics world
}
```

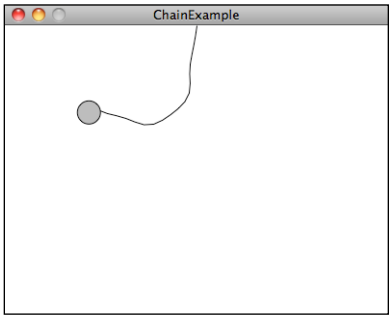
Now, what if we want the string to hang from a fixed point? We can lock one of the particles—the first, the last, the middle one, etc. Here's how we would access the first particle (in the ArrayList) and lock it.

```
Particle head=particles.get(0);
head.lock();
```

And if we want to draw all the particles connected with a line along with a circle for the last particle, we can use ***beginShape()***, ***endShape()***, and ***vertex()***, accessing the particle locations from our ArrayList.

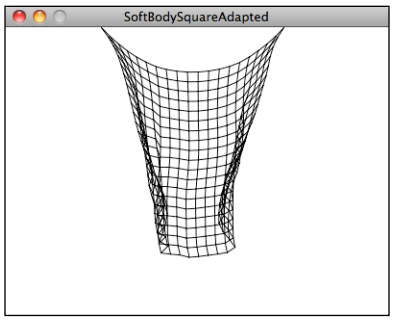
Example 5.x: Soft Swinging Pendulum

```
stroke(0);
noFill();
beginShape();
for (Particle p : particles) {
  vertex(p.x,p.y);     $$ Each particle is one point in the line
}
endShape();
Particle tail = particles.get(numPoints-1);
tail.display();  $$ This draws the last particle as a circle
```



The full code available with the chapter download also demonstrates how to drag the tail particle with the mouse.

Exercise: Create a hanging cloth simulation using the technique above, but connect all the particles with a grid as demonstrated in the screenshot below.



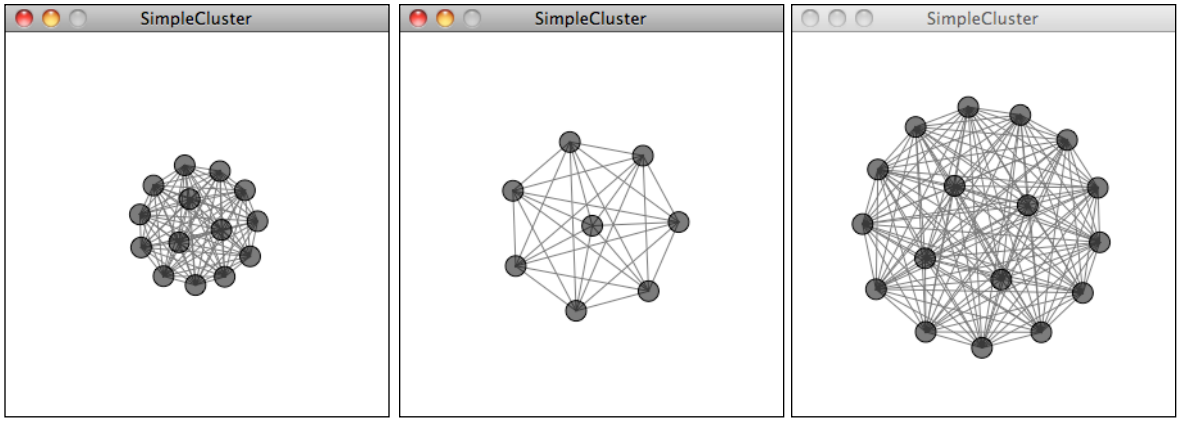
5.21 Connected Systems Part II: Force Directed Graph

Have you ever encountered the following scenario?

"I have a whole bunch of stuff I want to draw on the screen and I want all that stuff to be spaced out evenly in a nice, neat, organized manner. Otherwise I have trouble sleeping at night."

This is not an uncommon problem in computational design. One solution is typically referred to as a "force-directed graph." A force-directed graph is a visualization of elements—let's call

them “nodes”—in which the positions of those nodes are not manually assigned. Rather, the nodes arrange themselves according to a set of forces. While any forces can be used, a typical example involves spring forces. And so toxiclibs is perfect for this scenario.



How do we implement the above?

First, we’ll need a Node class. This is the easy part; it can extend VerletParticle2D. Really, this is just what we did before, only we’re calling it Node now instead of Particle.

```
class Node extends VerletParticle2D {
  Node(Vec2D pos) {
    super(pos);
  }

  void display() {
    fill(0,150);
    stroke(0);
    ellipse(x,y,16,16);
  }
}
```

Next we can write a class called Cluster, which will describe a list of Nodes.

```
class Cluster {

  ArrayList<Node> nodes;

  float diameter;   $$ We'll use this variable for the rest length between all the nodes

  Cluster(int n, float d, Vec2D center) {
    nodes = new ArrayList<Node>();
    diameter = d;

    for (int i = 0; i < n; i++) {
      nodes.add(new Node(center.add(Vec2D.randomVector())));
                              $$ Here's a funny little detail.  We're going to have a problem
    }                         if all the Node objects start in exactly the same location.
  }                           So we add a random vector to the center location so that each
                              Node is slightly offset.
```

Let's assume we added a ***display()*** function to draw all the Node objects in the Cluster and then created a Cluster object ***setup()*** and displayed it in ***draw()***. If we ran the sketch as is, nothing would happen. Why? Because we forgot the whole force-directed graph part! We need to connect every single Node to every other Node with a force. But what exactly do we mean by that? Let's assume we have four Node objects: 0, 1, 2 and 3. Here are our connections:

```
0 connected to 1
0 connected to 2
0 connected to 3
1 connected to 2
1 connected to 3
2 connected to 3
```

Notice two important details about our connection list.

- ***No Node is connected to itself.*** *We don't have 0 connected to 0 or 1 connected to 1.*
- ***We don't need to repeat connections in reverse.*** *In other words, if we've already said 0 is connected to 1, we don't need to say 1 is connected to 0 because, well, it already is!*

So how do we write code to make these connections for N number of nodes?

Look at the left column. It reads: 000 11 22. So we know we need to access each Node in the list from 0 to N-1.

```
for (int i = 0; i < nodes.size()-1; i++) {
  VerletParticle2D ni = nodes.get(i);
```

Now, we know we need to connect Node 0 to Nodes 1,2,3. For Node 1: 2,3. For Node 2: 3. So for every Node i, we must loop from i+1 until the end of the list.

```
          $$ Look how we start j at i + 1
  for (int j = i+1; j < nodes.size(); j++) {
    VerletParticle2D nj = nodes.get(j);
```

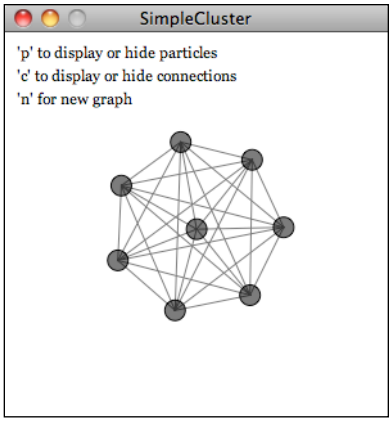
With every two Nodes we find, all we have to do then is make a VerletSpring2D.

```
                                    $$ The Spring connects Nodes "ni" and "nj"
    physics.addSpring(new VerletSpring2D(ni,nj,diameter,0.01));
  }
}
```

Assuming those connections are made in the Cluster constructor, we can now create a Cluster in our main tab and see the results!

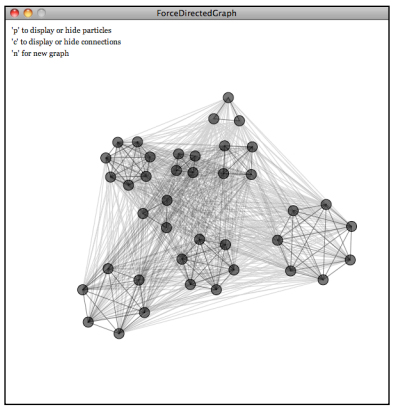
Example 5.x: Cluster

```
import toxi.geom.*;
import toxi.physics2d.*;

VerletPhysics2D physics;
Cluster cluster;

void setup() {
  size(300,300);
  physics=new VerletPhysics2D();
  cluster = new Cluster(8,100,new Vec2D(width/2,height/2));
      $$ Make a cluster
}

void draw() {
  physics.update();
  background(255);
  cluster.display();          $$ Draw the Cluster
}
```



Exercise: Use the Cluster structure as a skeleton for a cute, cuddly, squishy creature (à la "Nokia Friends"). Add gravity and also allow the creature to be dragged with the mouse.

Exercise: Expand the Force Directed Graph to have more than one Cluster object. Use a VerletMinDistanceSpring2D to connect Cluster to Cluster.



5.22 Attraction and Repulsion Behaviors

When we looked at adding an attraction force to Box2D, we found that the Box2D Body object included an ***applyForce()*** function. All we needed to do was calculate the attraction force (Force = G * mass1 * mass2 / distance squared) as a vector and apply it to the Body. toxiclibs also includes a function called ***addForce()*** that we can use to apply any calculated force to a VerletParticle.

However, toxiclibs also takes this idea one step further by allowing us to attach some common forces (let's call them "Behaviors") to VerletParticles, calculating them and applying them for us!

For example, if we attach an AttractionBehavior to a VerletParticle, then all other particles in the physics world will be attracted to that particle.

Let's say we have a Particle object (which extends VerletParticle).

```
Particle p = new Particle(new Vec2D(200,200));
```

Once we have that Particle, we can create an AttractionBehavior object associated with that Particle.

```
float distance = 20;
float strength = 0.1;
AttractionBehavior behavior = new AttractionBehavior(p, distance, strength);
```

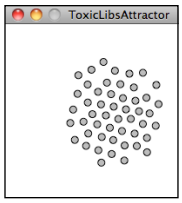
Notice how the behavior is created with two parameters—distance and strength. The distance specifies the range within which the behavior will be applied. For example, in the above scenario, only other Particle objects within 20 pixels will feel the Attraction force. The strength, of course, specifies how strong the force is.

Finally, in order for the force to be activated, the behavior needs to be added to the physics world.

```
physics.addBehavior(behavior);
```

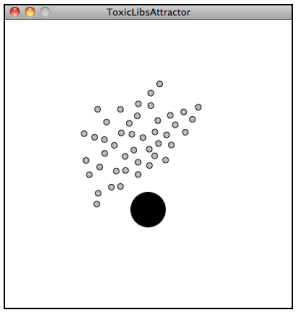
This means everything that lives in the physics simulation will always be attracted to that Particle object, as long as it is within the distance threshold.

Even though toxiclibs does not handle collisions, you can create a collision-like effect by adding a repulsive behavior to each and every Particle (so that every Particle repels every other Particle). Let's look at how we might modify our Particle class to do this.




```
class Particle extends VerletParticle2D {

  float r;       $$ We've added a radius to every Particle

  Particle (Vec2D loc) {
    super(loc);
    r = 4;
    physics.addBehavior(new AttractionBehavior(this, r*4, -1));
  }              $$ Every time a Particle is made, an AttractionBehavior is
                 generated and added to the physics world.  Note that when the strength
                 is negative, it's a repulsive force!

  void display () {
    fill (255);
    stroke (255);
    ellipse (x, y, r*2, r*2);
  }
}
```

We could now recreate our Attraction example by having a single Attractor object that exerts an AttractionBehavior over the entire window.




```
Example 5.x: Attraction / Repulsion
class Attractor extends VerletParticle2D {

  float r;

  Attractor (Vec2D loc) {
    super (loc);
    r = 24;
    physics.addBehavior(new AttractionBehavior(this, width, 0.1));
                        $$ The AttractionBehavior "distance" equals
                        the width so that it covers the entire window.
  }

  void display () {
    fill(0);
    ellipse (x, y, r*2, r*2);
  }
}
```

Exercise: Create an object that both attracts and repels. What if it attracts any Particle that are far away but repels those Particles at a short distance?

Exercise: Use AttractionBehavior in conjunction with Spring forces.

The Eco-System Project:

Step 5 Exercise:

Take your system of creatures from Step 4 and use a physics engine to drive their motion and behaviors. Some possibilities:

- *Use Box2D to allow collisions between creatures. Consider triggering events when creatures collide.*
- *Use Box2D to augment the design of your creatures. Build a skeleton with distance joints or make appendages with revolute joints?*
- *Use toxiclibs to augment the design of your creature. Use a chain of toxiclibs particles for tentacles or a mesh of springs as a skeleton.*
- *Use toxiclibs to add attraction and repulsion behaviors to your creatures.*
- *Use spring (or joint) connections between objects to control their interactions. What if you create and delete these springs on the fly? Consider making these connections visible or invisible to the viewer.*