

Creating Video Games and Video Game Controllers with Analog Pong and Processing

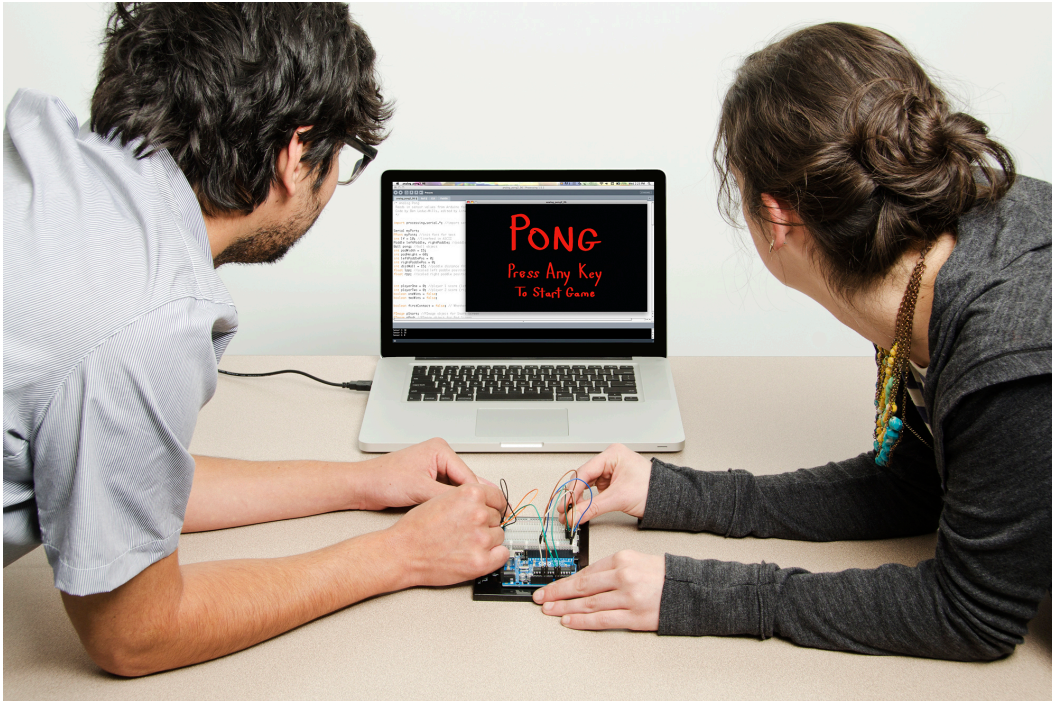


Photo by Juan Peña, sweet gaming by SparkFun's Customer Service

Learn how to create a video game and controller while learning basic programming, Algebra, Trigonometry, Cartesian graphing methods, Serial communication and analog sensor input. The skills and concepts covered in this activity are a great introduction to the same skills you would need to create embedded systems, graphical user interfaces and data logging technology, as well as work with robotics, GPS and more. But this way you're creating a game that you can play with your friends in the process!

Goals:

1. To get kids excited about Science, Technology, Engineering and Mathematics through game and game controller creation.
2. Establish baseline computer programming skills including: Commenting, constructive file saving methods, using variables, parsing variables out of a delimited string (don't get scared on me now, this one is simpler than you think), nested "if" statements, passing arguments to object instances (ok, this sounds hard if you've never programmed before, but it's not), creating functions and interactive graphics culminating in an individualized game that the students can play.
3. Create two simple analog sensor circuits using voltage dividers as the source of signal input. (I admit, there's some electrical engineer talk in here, but that's good, right? Plus, it's fun and easy!)
4. Establish Serial Communication between two computers and parse out data values from the communication. This skill is invaluable in all wireless communication including GPS signal manipulation and, like I said, it's more painless than you think.
5. Simulate very basic ball physics through the use of algebra, the Cartesian coordinate system and trigonometry.
6. Add images to the game and fit the image size to the screen size.
7. Use an integer variable counter to track time and control game speed with the counter.

Common Core Math Standards:

- 6.RP - Understand the concept of a ratio and use ratio language to describe a ratio relationship between two quantities.
- 6.NS - Understand that positive and negative numbers are used together to describe quantities having opposite directions or values (e.g., temperature above/below zero, elevation above/below sea level, credits/debits, positive/negative electric charge); use positive and negative numbers to represent quantities in real-world contexts, explaining the meaning of 0 in each situation.
- 6.NS - Understand a rational number as a point on the number line. Extend number line diagrams and coordinate axes familiar from previous grades to represent points on the line and in the plane with negative number coordinates.
- 6.NS - Understand ordering and absolute value of rational numbers.
- 6.NS - Solve real-world and mathematical problems by graphing points in all four quadrants of the coordinate plane. Include use of coordinates and absolute value to find distances between points with the same first coordinate or the same second coordinate.
- 6.EE - Write and evaluate numerical expressions involving whole-number exponents.
- 6.EE - Write, read, and evaluate expressions in which letters stand for numbers.
- 6.EE - Evaluate expressions at specific values of their variables. Include expressions that arise from formulas used in real-world problems.
- 6.EE - Understand solving an equation or inequality as a process of answering a question: which values from a specified set, if any, make the equation or inequality true?
- 6.EE - Use variables to represent numbers and write expressions when solving a real-world or mathematical problem; understand that a variable can represent an unknown number, or, depending on the purpose at hand, any number in a specified set.
- 6.EE - Solve real-world and mathematical problems by writing and solving equations of the form $x + p = q$ and $px = q$ for cases in which p , q and x are all nonnegative rational numbers.
- 6.EE - Write an inequality of the form $x > c$ or $x < c$ to represent a constraint or condition in a real-world or mathematical problem. Recognize that inequalities of the form $x > c$ or $x < c$ have infinitely many solutions; represent solutions of such inequalities on number line diagrams.
- 6.EE - Use variables to represent two quantities in a real-world problem that change in relationship to one another; write an equation to express one quantity, thought of as the dependent variable, in terms of the other quantity, thought of as the independent variable. Analyze the relationship between the dependent and independent variables using graphs and tables, and relate these to the equation.
- 7.RP - Compute unit rates associated with ratios of fractions, including ratios of lengths, areas and other quantities measured in like or different units.
- 7.RP - Recognize and represent proportional relationships between quantities.
- 7.RP - Use proportional relationships to solve multistep ratio and percent problems.
- 7.NS - Apply and extend previous understandings of addition and subtraction to add and subtract rational numbers; represent addition and subtraction on a horizontal or vertical number line diagram.
- 7.NS - Apply and extend previous understandings of multiplication and division and of fractions to multiply and divide rational numbers.
- 7.NS - Solve real-world and mathematical problems involving the four operations with rational numbers.
- 7.EE - Understand that rewriting an expression in different forms in a problem context can shed light on the problem and how the quantities in it are related.

7.EE - Solve multi-step real-life and mathematical problems posed with positive and negative rational numbers in any form (whole numbers, fractions, and decimals), using tools strategically. Apply properties of operations to calculate with numbers in any form; convert between forms as appropriate; and assess the reasonableness of answers using mental computation and estimation strategies.

7.G - Know the formulas for the area and circumference of a circle and use them to solve problems; give an informal derivation of the relationship between the circumference and area of a circle.

7.G - Use facts about supplementary, complementary, vertical, and adjacent angles in a multi-step problem to write and solve simple equations for an unknown angle in a figure.

8.EE - Solve linear equations in one variable.

8.F - Understand that a function is a rule that assigns to each input exactly one output. The graph of a function is the set of ordered pairs consisting of an input and the corresponding output.

8.F - Construct a function to model a linear relationship between two quantities. Determine the rate of change and initial value of the function from a description of a relationship or from two (x, y) values, including reading these from a table or from a graph. Interpret the rate of change and initial value of a linear function in terms of the situation it models, and in terms of its graph or a table of values.

8.G - Describe the effect of dilations, translations, rotations, and reflections on two-dimensional figures using coordinates.

8.G - Explain a proof of the Pythagorean Theorem and its converse.

8.G - Apply the Pythagorean Theorem to determine unknown side lengths in right triangles in real-world and mathematical problems in two and three dimensions.

8.G - Apply the Pythagorean Theorem to find the distance between two points in a coordinate system.

HSN-Q - Use units as a way to understand problems and to guide the solution of multi-step problems; choose and interpret units consistently in formulas; choose and interpret the scale and the origin in graphs and data displays.

HSN-VM - Solve problems involving velocity and other quantities that can be represented by vectors.

HSN-VM - Add and subtract vectors.

HSA-CED - Create equations and inequalities in one variable and use them to solve problems. *Include equations arising from linear and quadratic functions, and simple rational and exponential functions.*

HSA-CED - Create equations in two or more variables to represent relationships between quantities; graph equations on coordinate axes with labels and scales.

HSA-CED - Represent constraints by equations or inequalities, and by systems of equations and/or inequalities, and interpret solutions as viable or nonviable options in a modeling context. *For example, represent inequalities describing nutritional and cost constraints on combinations of different foods.*

HSA-CED - Rearrange formulas to highlight a quantity of interest, using the same reasoning as in solving equations. *For example, rearrange Ohm's law $V = IR$ to highlight resistance R .*

HSA-REI - Explain each step in solving a simple equation as following from the equality of numbers asserted at the previous step, starting from the assumption that the original equation has a solution. Construct a viable argument to justify a solution method.

HSA-REI - Solve linear equations and inequalities in one variable, including equations with coefficients represented by letters.

HSF-IF - Understand that a function from one set (called the domain) to another set (called the range) assigns to each element of the domain exactly one element of the range. If f is a function and x is an element of its domain, then $f(x)$ denotes the output of f corresponding to the input x . The graph of f is the graph of the equation $y = f(x)$.

HSF-BF - Write a function that describes a relationship between two quantities.

HSF-BF - Write arithmetic and geometric sequences both recursively and with an explicit formula, use them to model situations, and translate between the two forms.

HSF-LE - Recognize situations in which one quantity changes at a constant rate per unit interval relative to another.

HSF-LE - Interpret the parameters in a linear or exponential function in terms of a context.

HSF-TF - Understand radian measure of an angle as the length of the arc on the unit circle subtended by the angle.

HSF-TF - Explain how the unit circle in the coordinate plane enables the extension of trigonometric functions to all real numbers, interpreted as radian measures of angles traversed counterclockwise around the unit circle.

HSF-TF - Use inverse functions to solve trigonometric equations that arise in modeling contexts; evaluate the solutions using technology, and interpret them in terms of the context.

HSG-SRT - Use trigonometric ratios and the Pythagorean Theorem to solve right triangles in applied problems.

HSG-C - Derive using similarity the fact that the length of the arc intercepted by an angle is proportional to the radius, and define the radian measure of the angle as the constant of proportionality; derive the formula for the area of a sector.

HSG-MG - Use geometric shapes, their measures, and their properties to describe objects (e.g., modeling a tree trunk or a human torso as a cylinder).

HSG-MG - Apply geometric methods to solve design problems (e.g., designing an object or structure to satisfy physical constraints or minimize cost; working with typographic grid systems based on ratios).

Materials:

1. A computer with Arduino and Processing (32 Bit version) installed
2. An Arduino compatible microcontroller
3. A breadboard (sometimes called a prototyping board)
4. Two potentiometers (this is just a fancy name for a dial). If you don't have two potentiometers, students can use any combination of two analog sensors intended for use with the Arduino system (I will be showing how to use a light sensor with a 10K Ω resistor as well).
5. Six jumper wires (any old wire will work as long as it can be plugged into the breadboard). This number may vary depending on the analog sensor. Some analog sensors may also require additional resistors.
6. For extra challenges - two buttons, two 10K Ω resistors (resistors are optional), an additional potentiometer or analog sensor and at least nine jumper wires.
7. The Serial drivers, so Processing and Arduino can talk to each other (this is covered later in this document).
8. And of course the Arduino and Processing Pong Code, available in a .zip file at <http://learn.sparkfun.com/curriculum/62>.

The Arduino and Processing files you will find at the link above and a little bit about these files:

analog_pong01.ino – Arduino code for use with a single analog sensor

analog_pong02.ino – Arduino code for use with two analog sensors (use analog_pong02.ino if you just want Pong up and running)

analog_pong2_01.pde – Processing code for working through this handout (non-functional without additional code)

analog_pong2_02.pde – Processing code after working through this handout (use analog_pong2_02.pde if you just want Pong up and running)

Note - All the parts necessary to complete this activity, with the exception of the computer and code, are included in the SparkFun Inventor's Kit.

<https://www.sparkfun.com/products/12001>

Suggested Questions Prior to Activity:

1. When was the first computer invented? What did the interface look like?
2. What was the first arcade game ever invented and when was it invented?
3. Why do you think it took so long for someone to make a game for computers?
4. What is your favorite computer or video game? What type of game is it? Is it a strategy game, a side scroller, a first-person shooter, a role-playing game, or maybe a 3-D puzzle?

Pong, with Arduino and Processing:

You will be making Pong, which was invented by Allan Alcorn in 1972 for a company called Atari. You will be using two software platforms called Arduino and Processing to write code. You will also be creating your own hardware controller interface for Pong so you can use dials, buttons and sensors instead of just your keyboard to control the game. Before we get started here's a little background, some help with installing Processing and general helpful hints about Processing (if you're already installed and ready to go, skip to page 10):

What Processing is:

Processing is a Java-based programming environment that draws on PostScript and OpenGL for 2-D and 3-D graphics, respectively.

Processing is a wonderful entry-level program that interfaces easily with Arduino via Serial, making it a simple yet powerful environment.

Who created Processing:

Processing was conceived at MIT in 2001 by Casey Reas and Ben Fry. Processing is a FLOSS project (Free, Libre, Open Source Software) with millions of contributors all linked by the Processing website, Processing.org. Processing has a system of software extensions called "libraries". This allows people to write code and extend the abilities of the original software for various purposes. These "libraries," including the Arduino library, which is just one way to interface Arduino hardware with your Processing sketches, are available on the website.

Downloading and installing Processing:

Go to <http://processing.org/download> and select Linux, Mac or Windows depending on what kind of machine you have.

For Linux:

Download the .tar.gz file to your home directory, then open a terminal window and type:

```
Tar xvfz processing-xxxx.tgz
```

(replace xxxx with the rest of the file's name, which is the version number)

This will create a folder named processing-1.5 or something similar. Then change to that directory:

```
cd processing-xxxx
```

and run processing:

```
./processing
```

For Mac:

Double-click the .dmg file and drag the Processing icon from inside this file to your applications folder, or any other location on your computer. Double click the Processing icon to start Processing.

For Windows:

Double-click the .zip file and drag the folder inside labeled Processing to a location on your hard drive. Double click the Processing icon to start Processing.

If you are stuck, go to <http://wiki.processing.org/index.php/Troubleshooting> for help.

9. In order to get Arduino and Processing talking to each other you will need to install a library and possibly take a couple extra steps beyond that. Luckily you can use this handy guide to help you out:

<http://www.arduino.cc/playground/interfacing/processing>

Make sure you or your IT people install this library before starting

this activity! You will want to make sure the changes allowed Serial communication between Arduino and Processing. In order to do, this load the file named “analog_pong01.ino” onto your Arduino board, close Arduino and run the Processing sketch called “analog_pong2_01.pde.” If you do not encounter an error message you are good to go.

You may also need to go to the website below for additional help if you get the following error message:

```
WARNING: RXTX Version mismatch
Jar version          = RXTX-2.2pre1
native lib Version   = RXTX-2.2pre2
```

<http://www.sundh.com/blog/2011/05/get-processing-and-arduino-to-talk/>

PROCESSING CHEAT SHEET

DATA TYPES

Primitive
 boolean
 byte
 char
 color
 double
 float
 int
 long

Composite
 Array
 ArrayList
 HashMap
 Object
 String
 XMLElement

Conversion

binary()
 boolean()
 byte()
 char()
 float()
 hex()
 int()
 str()
 unary()
 unhex()

String Functions

join()
 match()
 matchAll()
 nf()
 nfc()
 nfp()
 nfs()
 split()
 splitTokens()
 trim()

Array Functions

append()
 arrayCopy()
 concat()
 expand()
 reverse()
 shorten()
 sort()
 splice()
 subset()

Constants

HALF_PI
 PI
 QUARTER_PI
 TWO_PI

Assign variables

```
=          assign value to a variable
;          statement terminator
,          separates parameters in function
           separates variables in declarations
           separates variables in array

/** Assign variables */
//Format is in variable_type variable_name;
int total;

//Then you can assign a value to it later
total = 0;

//Or, assign a value to it at the same time
int total = 0;

//Note: use one of the primitive data types
on the left
```

Structure: program structure

```
setup()    defines initial environment
           properties, screen size,
           background before the draw()

draw()     called after setup() & executes
           code continuously inside its
           block until program is stopped
           or noLoop() is called.

size()     size() must be first line in
           setup() defines dimension of
           display in units of pixels

noLoop()   Stops Processing from executing
           code within draw()
           continuously
```

```
/** Example */
void setup() {
  size(200, 200);
  background(0);
  fill(102);
}

void draw() {
  //Draw code here
}
```

2D Primitives

```
point()    draws a point
           point(x, y)
           point(x, y, z) //3D

line()     draws a line
           line(x1, y1, x2, y2)
           line(x1, y1, z1, x2, y2, z2) //3D

rect()     draws a rectangle
           rect(x, y, width, height)

ellipse()  Draws an ellipse
           ellipse(x, y, width, height)

arc()      draws an arc
           arc(x, y, width, height, start, stop)

/** Arc (portion of circle) */
//x & y = coords, width & height = size
//start + stop = starting and end points
(think angle in radians) of circle in π pie
```

[LINK](#)

```
arc(x, y, width, height, start, stop)
arc(100, 100, 50, 50, PI, 2*PI) //Sad Face
arc(100, 100, 50, 50, 0, PI) //Happy Face
//Note: Play around with start and stop. Use
PIE constants or math operators PI/3, .5*PI
```

Relational

```
==         equality
>          greater than
>=         greater than or equal to
!=         inequality
<=         less than or equal to

/** Example */
if(total == 100){
  //Then do this
}
```

Iteration

```
while      executes statements while the
           expression is true

for        loop continues until the test
           evaluates to false

/** while Example */
while(total < 100){
  total++; //adds 1 to total
}

/** for Example */
for(int i=0; i<100; i++){
  //Do something here
}
```

Conditionals

```
if         if statement evaluates to true
           then execute code

else       extension of if statement
           executes if equals false

else if    extension of if statement
           executes if equals true

/** if / else / else if */
if(total == 100){
  //total is equal to 100
}
else
if(total < 100){
  //total is smaller than 100
}
else{
  //total is bigger than 100
}
```

Coloring stuff

```
background() sets background color in RGB or
             hexadecimal color
             background(value1, value2,
             value3)
             background(hexadecimal_value)

fill()       sets color for shape
             fill(value1, value2, value3)
             fill(hexadecimal_value)

stroke()     sets color for shape
             stroke(value1, value2, value3)
             stroke(hexadecimal_value)

/** Example */
//Note call fill or stroke before every shape you
are planning on using different colors on each
stroke(CCCFFF);
fill(FFCCCC);
rect(100, 100, 50, 50);
```

CONTROL

Relational Operators

```
== (equality)
> (greater than)
>= (greater than or
    equal to)
!= (inequality)
< (less than)
<= (less than or equal
    to)
```

Iteration

```
for
while
```

Conditionals

```
break
case
?: (conditional)
continue
default
else
if
switch()
```

Logical Operators

```
&& (logical AND)
! (logical NOT)
|| (logical OR)
```

Cheat Sheet courtesy of Chrisdrogaris.com

A few things before we start:

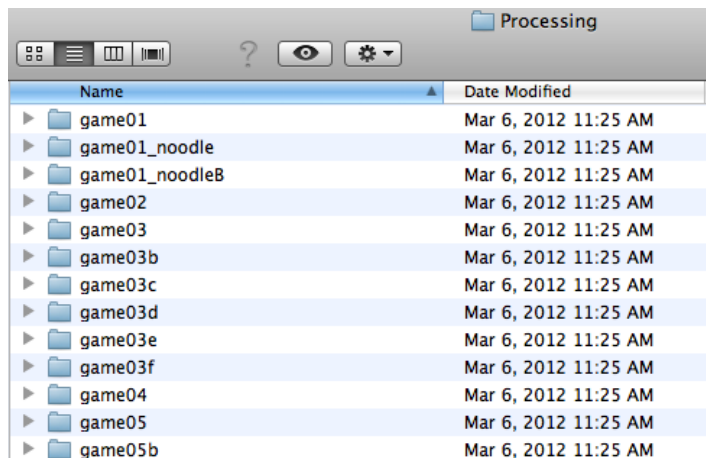
Feel free to make the game more exciting or interactive at any time by adding your own code. Just make sure of two things so you and your teacher don't get confused:

1. Make sure you add comments whenever you add your own code. Try to explain what you want the code to do, or go back and explain exactly what the code does once you finish it. Comments do not affect code.

```
//This is a comment I can put it here
void setup () { //or I can put it here

}
```

2. Make sure you save a different version each time you save. For example maybe my first file is called Pong01.pde. The second time I save it I would call it Pong02.pde. If I'm trying to change some colors in the game I might want to call it Pong02_ColorChange.pde the next time I save it. This way if I mess up my code I can always go back to a previous version and see what I changed. Use descriptive names so you have an easier time hunting through the different files as you build your game.



Example of file naming format in my Processing saved sketches folder

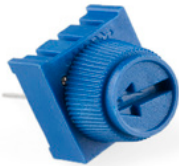
Before we really dive into code we're going to put the circuits together in the next section.

Creating Pong and the Game-Controlling Circuits:

Creating the Circuits:

1. First thing's first, let's make the two simple circuits that you will use to control your game. You will be plugging these circuits into your breadboard, but before we do that we need to find the parts that make the circuits. There are a bunch of different circuits you can create to control Pong. I have outlined two below, but you can use just about any analog sensors that operate in a 0-5 volt range. The sensors I outline below are a potentiometer (dial) and a light sensor. Here are the parts you will need to put together each of the circuits:

For Potentiometer
Circuit:



One potentiometer
and five jumper
wires

For Light Sensor
Circuit:



One 10K Ω
resistor, one
light sensor
and five
jumper wires

For both Circuits:



Jumper wires

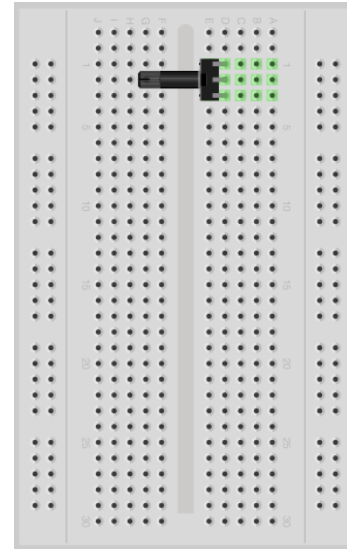
(You could do this with
three wires for each
circuit but this way is a
little less confusing.)

2. Now find your breadboard.
It may look like this:
Don't worry if it looks a little
different. The important thing
is that you can plug wires
and components into your
breadboard.

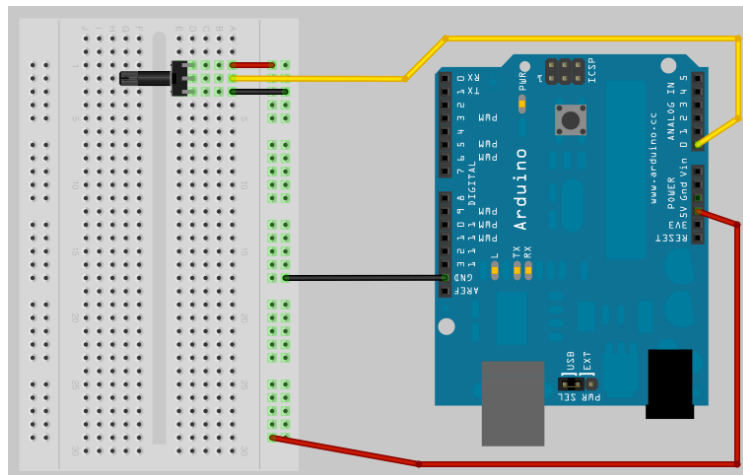
For more info on breadboards, go to
<http://www.sparkfun.com/tutorials/202>



3. First plug the potentiometer (dial) into the breadboard. Make sure that the “leads” (the wires sticking out of the potentiometer) are sticking into the breadboard up and down, not sideways. This means that the leads will be stuck into three different rows and one single column. The leads will not be stuck into three different columns and just one row. If you’re having trouble understanding this, check out the image to the right.

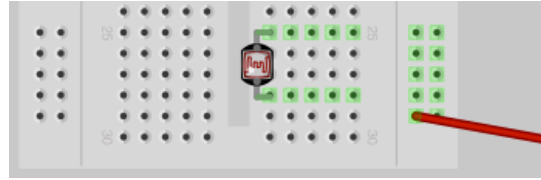


4. Next, plug the breadboard and potentiometer into your Arduino following the image to the right. You can use any color wire, but it is general practice to use a red wire for any power connections and a black (or similar dark color) wire for any ground connections. Use any other color for the

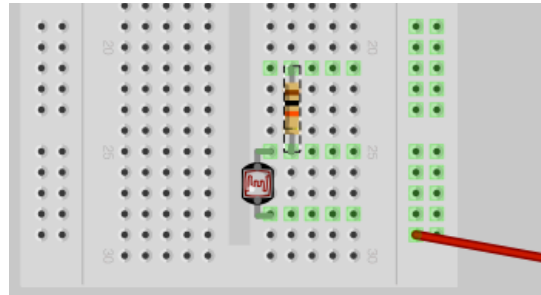


communication line. Often engineers will use different color com lines for each sensor to make it easier to tell the lines and sensor circuits apart. The communication lines are the lines that send electrical values to the Arduino microcontroller. In this case I used yellow for my communication line. Make sure that the top lead on the component has a connection to the 5V pin on the Arduino, a connection to Arduino A0 in the middle and a connection from the bottom lead to the Arduino GND pin. The potentiometer is not polarized, so you can actually switch the power wire so it is connected to the bottom of the potentiometer, as long as you switch the ground wire so it is connected to the top instead of the bottom as well.

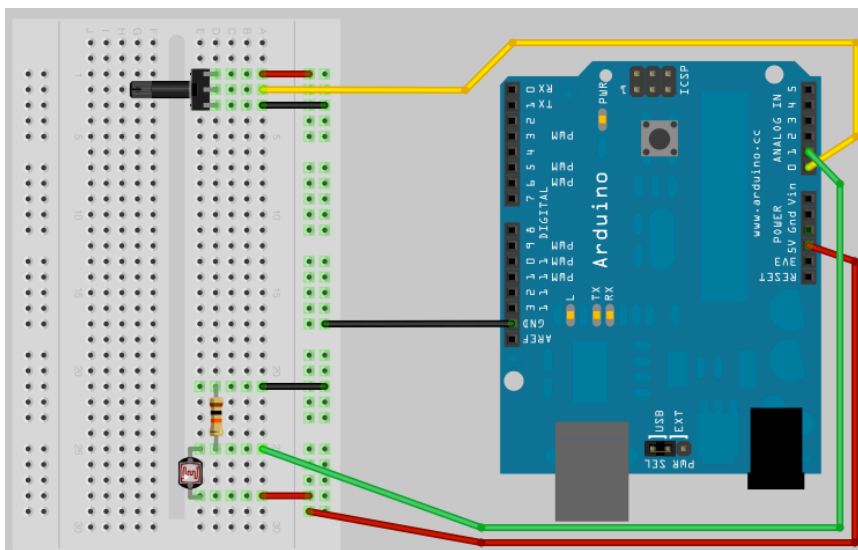
5. Ok. That's one player's controller plugged in. If you have two potentiometers feel free to use two of them, but this next sensor is a little trickier to plug in so I've broken it down into a couple steps. First plug the light sensor into your breadboard. Just like the potentiometer, make sure you plug it in so the leads are stuck into two different rows. To save space I've only shown the bottom of the breadboard, where I plugged in my light sensor.



6. Next plug the 10K Ω resistor into the breadboard. Make sure that one side of the resistor is plugged into the same row as the upper light resistor lead. The other side of the resistor should plug into the breadboard even further up on the breadboard, away from the light sensor. Make sure you use the 10K Ω resistor and not a 330 Ω resistor!



7. Now plug this sensor into the Arduino microcontroller using a power wire (5V connection), a communication wire and a ground wire (GND connection). This should look very similar to the way you plugged in the potentiometer before. There are a few differences though; the communication wire should plug into Arduino pin A1, since we've already used pin A0 to plug in the potentiometer. Also this sensor circuit is polarized, so you need to make sure that the power is connected to the bottom lead of the light sensor and that the ground is connected to the top lead of the resistor.



8. Great! You've built the circuits you need to control a simple two-player game. Next we'll talk about how these analog sensor circuits work. If you want to get to the coding portion of this activity, feel free to skip to the next section and come back later to learn about voltage dividers.

For additional help with creating these two circuits visit:

<http://learn.sparkfun.com/products/2>

And click on "Web quality guide" in the Documentation section:

Documentation:

- [Web quality guide \(1MB\)](#)
- [Print quality guide \(9MB\)](#)
- [Circuit overlays](#)
- [SIK Review](#)
- [SIK Education Binder](#)
- [Parts Wishlist](#)
- [Example Code](#)

Creating the Circuits - Voltage Dividers:

1. In this section we will be discussing variable resistance analog sensors and how they work. Specifically I will outline how the potentiometer, photo-resistor (light sensor) and flex sensor operate.

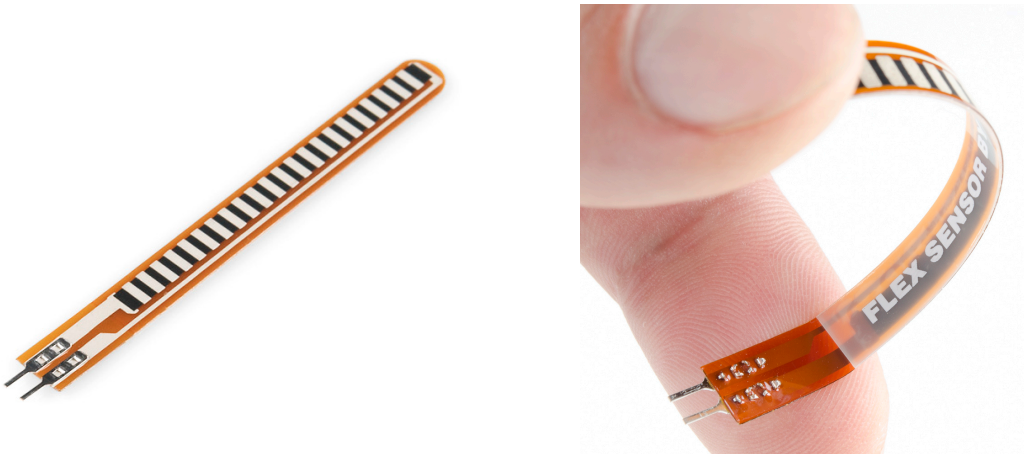
The sensors you are using operate through a property called variable resistance. This means that the electrical signal the Arduino receives from the sensor depends on the resistance of a piece in the sensor. Depending on the sensor, the resistance of that piece in turn depends on how far you turn the knob (for a potentiometer), how much sunlight is hitting the sensor or, in the case of a flex sensor, how far you are bending the sensor. The action that activates the sensor causes the resistance inside the sensor to vary, which is why it is called variable resistance.



The resistance depends on how much you turn the dial



There is less resistance when less light hits the light sensor, more resistance when more light hits the sensor



There is less resistance when the flex sensor is straight and more resistance the more the flex sensor is bent (note that this sensor only bends in one direction)

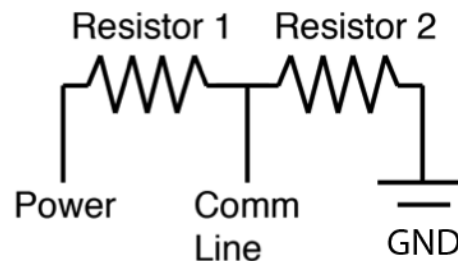
2. For each of the analog sensors you are using there are three connections which are standard for all variable resistance analog sensors. These three connections are power (in this case 5V), the communication line and ground. For some sensors, such as the photo-resistor and flex sensor, you actually need to add a resistor (in this case you would use 10K resistors) to the sensor circuit in order to create all three of those connections (More on the additional resistor later). The reason for these three connections is that all circuits need a connection to power and ground to allow electricity to flow through the components in the circuit, and all sensor circuits need a communication wire off of which you can pull readings. Without a communication line, how would you ever know what the sensor's value is?

3. The analog sensor's communication line has two resistors on either side (one closer to power and one closer to ground) in the electrical circuit. One or two of these resistors is a variable resistor. In the case of the potentiometer, both of the resistors are variable resistors, but for the photoresistor and flex sensor the variable resistor is in the sensor, and the resistance value of the 10K resistor stays the same.

Let's examine how the electricity behaves when it flows through a simple voltage divider circuit. First, the electricity enters the circuit from the power connection on the Arduino; in this case these values are 5V and 200 mA. The first thing the electricity encounters is the variable resistance element of the sensor. This impedes the ability of the electricity a little, but the resistance of the sensor should never be enough to stop the flow of current. Then the electricity has the option to keep traveling through the circuit towards ground, or the electricity can travel along the communication line back to the Arduino analog input pin. A portion of the electricity does both of these things unless the sensor is either maxed out or not reading any data at all, in which case all the electricity either goes to the analog input pin, or it continues through the circuit to ground. The electricity that continues through the sensor circuit and travels to ground meets a little additional resistance from the second resistor in the circuit. In the case of the photoresistor and flex sensor, you can actually see the second resistor through which the electricity must travel.

Because there is resistance to both paths the electricity can travel, the entirety of the electricity does not take one path or the other. Instead, it takes both. (Unless the sensor is maxed out, in which case the electricity does just take one path or the other.) Depending on the ratio of resistance between the two resistors in the circuit, more or less electricity travels through the communication line instead of the ground. The circuit effectively divides the voltage by supplying two paths it can take, that's why it's called a "voltage divider." This is how we get a reading off of most analog sensors attached to the Arduino.

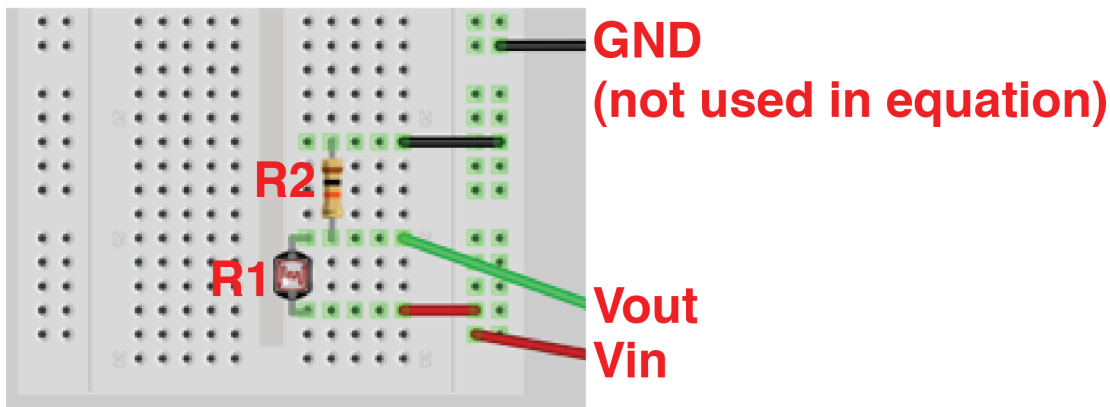
As Resistor 1's resistance increases, more of the electricity is converted to heat and escapes into the air, instead of traveling along the circuit to the comm line and ground.



4. There is an equation to help you figure out the two resistance values and resulting voltage value. I know you just can't wait to get your hands on that equation and plug some values in, so here it is:

$$V_{out} = V_{in} * (R2/(R1+R2))$$

Let's break down those variables: V_{out} is the amount of voltage that we read off of the communication line of the voltage divider (the whole point of our voltage dividers). V_{in} is the amount of voltage you are supplying to the circuit with your power connection. $R1$ is the first resistor value that the electricity encounters when it is traveling from power to ground. $R2$ is the second resistor value in the circuit; this is the resistor that is closer to ground after the communication line connection.



Example of light sensor voltage divider circuit with equation variables labeled.

Writing the Game Code:

Writing the Arduino code:

1. Open the file named “Analog_Pong01.ino.” The first two steps we will take are creating a comment and saving the file with a different name so you know it’s yours. Comments are important because they help you figure out what you were thinking when you wrote your code, keep track of variables, take credit for your work, and help other people use your code. Programmers often use commenting as a way to keep code visible but keeping the code from affecting the program they are writing. If something is going wrong, often programmers will comment lines of code they suspect are creating errors until the error goes away. Then they know which line or lines of code were creating the error (also called bugs). Comments are displayed in Arduino as **gray text**, and they do not affect your code. There are two different ways to make comments; the simplest way is to write two slashes like this: `//`. This will allow you to make one line of comments. Go ahead and make a comment with your name in it just above the `setup` function. (The beginning of the `setup` function is the part of the code that reads `void setup () {`.)

```
analog_pong01 $
/*
  AnalogPong
  Reads in 1 sensor value as paddles to send to a processing sketch
  Code by Ben Leduc-Mills, edited by Linz Craig
  */
//Make your comment here
void setup() {
  Serial.begin(9600); //Establish rate of Serial communication
```

The other way to make comments is to create a block of comments (multiple lines) by typing `/*` at the beginning of the block of comments and then `*/` at the end. Everything between these lines will be commented and will not affect your code. It is common practice to create a block of comments at the top of your code to explain who wrote it and what it does. It’s also good to give credit to anyone else’s code that you copied and pasted; a lot of people borrow code it so it doesn’t make your code any less impressive.

Now go to the File menu tab and choose Save As. This will save your file with a different name. It's important to save versions of your code as you write it so that if you make a mistake you can go back and look at the code that did work and compare it to the code that doesn't work now. Find the folder where you will be saving your Arduino files and rename the file with your initials in it and as version 02. For example I might call my new file "Analog_Pong_LRC02"; don't worry about adding the .ino file extension to the end of the file name. The file extension .ino is so that the computer knows the file is an Arduino file and Arduino adds it onto the end of the files automatically.

2. Next we need to add a line of code that saves the value of the second analog sensors as an integer variable. There is a line of code that is almost exactly what we need already in the code. Feel free to copy and paste that code to the correct space. If you do this, make sure you change the copied code to save the value of the sensor on Arduino pin A1, instead of A0. Then make sure that this value is being save to a different variable, otherwise you will overwrite your old value and run into some confusing issues. Because the variable name "leftPaddle" is already taken, the most obvious choice for this second variable name would be "rightPaddle." The ", DEC" portion of the code simply tells Arduino to communicate this value as a base ten number, because that's how humans think.

Copy this line — `int leftPaddle = analogRead(A0); //save the va`
change two things `Serial.print(leftPaddle, DEC); //print out on :`
and paste it here — `Serial.print(","); //print out on :`
`//you have to save the value from analog senso`
`//print out on Serial the value of the second :`

- Now you've got a variable named something like "rightPaddle," but we need Arduino to print the line out over the Serial line so that your computer can receive it and use it in Processing. To do that we need to use a `Serial.println()` command. This is almost the same as the line of code used to print out the `leftPaddle` variable, but look closely: it's not quite the same. When you print out `rightPaddle` you will need to use `Serial.println` instead of `Serial.print`. This is because `Serial.println` sends a carriage return at the end of the serial communication. A carriage return is exactly like hitting the "return" button after typing the variable value. Why do we have to do this? We have to use `Serial.println` because the Processing sketch is going to be listening for that carriage return in order to know that all of the sensor values have been sent, and the next sensor value is going to be the first sensor all over again. So go ahead and copy the `Serial.print` line of code and paste it below the `rightPaddle` variable line. Make sure you change the "print" to "println" and make sure you're sending `rightPaddle` instead of `leftPaddle`.

Copy this line — `int leftPaddle = analogRead(A0); //save the va`
— `Serial.print(leftPaddle, DEC); //print out on :`
 change two things — `Serial.print(","); //print out on :`
 and paste it here — `//you have to save the value from analog senso`
— `//print out on Serial the value of the second :`

Here's my resulting code from the last two steps:

```
int leftPaddle = analogRead(A0); //save
Serial.print(leftPaddle, DEC); //print
Serial.print(","); //print
int rightPaddle = analogRead(A1); //sav
Serial.println(rightPaddle, DEC);
```

- Ok. You've changed a little bit of the code, so guess what? You need to save it. Make sure you change the number at the end of the file name to 03 instead of 02. After you've saved the file, plug your Arduino into your computer and upload the sketch to the Arduino.

Having trouble uploading? Make sure you've got the right board and com port selected. To check these go to Tools in the menu tab and select either "Board" to change the board type, or "Serial Port" to select your com port.

5. Now before we move on, open your Serial Monitor (found underneath the Tools tab in the menu) and watch the values of your variables change as you play with the sensors. What is the highest and lowest these numbers will reach? Write them down for future reference.

The rest of this section is dedicated to explaining the rest of the code on the Arduino. Feel free to skip ahead to the next section (Writing the Processing code) and come back to learn about Serial communication later.

Serial Communication Handshake, Arduino:

There are really only two parts to this handshake other than actually sending the data (which we already did using `Serial.print` and `Serial.println`). These two parts establish the Serial communication rate in the `setup` function, and make Arduino wait for a return signal from Processing so it knows when to start sending data.

The first portion establishes Serial communication at a Baud rate of 9600. Baud means “symbols per second” or “pulses per second.” The reason the Baud rate is important is because Arduino needs to communicate at the same rate that Processing listens, otherwise they won’t understand each other. The command that establishes Serial communication is `Serial.begin` with the Baud rate inside the parenthesis.

```
void setup() {  
  Serial.begin(9600); //Establish rate of Serial communication  
  establishContact(); //See function below  
}
```

You can use a different Baud rate (300, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200) but there are strengths and weaknesses to using a higher Baud rate. The higher the Baud rate, the quicker Processing will receive sensor data, but the more processing power it takes up. So it’s a give and take relationship; for most purposes at the introductory level a Baud rate of 9600 will work just fine.

The second part of the Arduino side of this Serial handshake is the `establishContact` function. The `establishContact` function is a `while` loop that keeps the Arduino from sending data until it has heard from Processing. To do this the Arduino uses the command `Serial.available()`. `Serial.available` checks the Arduino's Serial communication buffer (a buffer is where incoming or outgoing data is temporarily stored) to see if Processing has sent any data to the Arduino yet. If the Arduino has not heard anything from Processing it continues to send the data "hello". This is because, as you will see, Processing is waiting to hear "hello" from Arduino before it sends any data back to the Arduino. It's a little complicated, but it's kind of like when you get your friend's attention before actually starting a conversation. If you just started straight into your conversation while your friend wasn't listening then a lot of the data would be lost, right? It's the same with Serial communication Arduino and Processing.

```
void establishContact() {  
  while (Serial.available() <= 0) { //when Arduino receives a Serial message  
    Serial.println("hello"); // send a starting message  
    delay(300); //Wait 300 milliseconds  
  }  
}
```

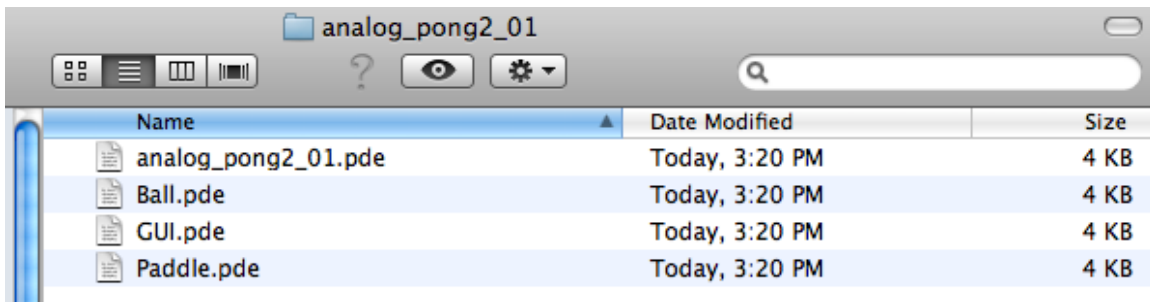
Once Arduino hears any Serial communication, the amount of information in the Serial buffer will be larger than zero. Then Arduino will exit the `while` loop and start sending sensor data.

So that's one side of Serial communication, this is a really simple example, but it represents the basic code you need to send data back and forth between Arduino and Processing. There are other, more complicated "handshakes," but this is a great place to start. This will also work for many other Serial communication setups other than just Arduino and Processing.

Writing the Processing code:

First thing's first: make sure that you have closed Arduino. Processing will not receive the correct Serial data if Arduino's Serial Monitor is open. Sometimes you'll run into issues even if Arduino's Serial Monitor isn't open, so it's best to just close Arduino.

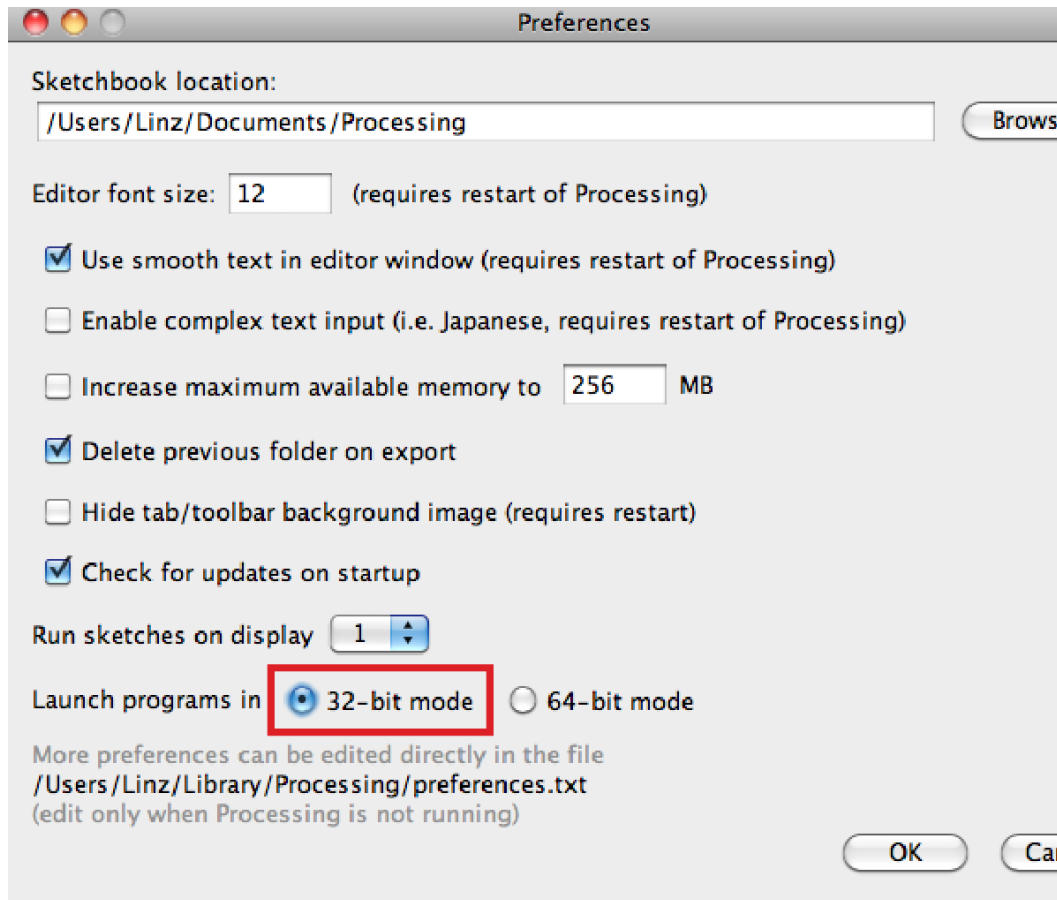
Ok. Now we are going to work on the Processing code. This is where the actual game will be displayed. To get you started we've included most of the code in the sketch already, but you will need to add some code to make it work, as well as some after that if you want to make the game better. To start, open the folder named `analog_pong2_01.pde`; it should look something like this:



Obviously it will look different if you are using a Windows or Linux machine.

The files you are looking at are the various parts of the Pong game. The file called `analog_pong2_01.pde` is the main file that contains everything else. The other files are created when a programmer makes a new tab inside of a Processing sketch. These files contain other “objects” that get used inside the main portion of code. Later on you will create an “instance” of the Ball “object.” For now just double click on the file called `analog_pong2_01.pde`. This will open the file in Processing. If you are having trouble opening the file make sure you have Processing (and the latest version of Java) installed. If you don't have Processing installed you can find it here: <http://processing.org/download/>.

Important! Make sure that you download the 32 bit version of Processing and not the 64 bit version. Serial Communication will not work with the 64 bit version. If you need to change your Processing version from 64 to 32 bit you can switch it inside of Preferences:



1. The first thing we have to do is add some code to make the Processing sketch “parse” out the data values that Processing received from the Arduino. In the `serialEvent` handler (the function that handles Serial events), the first thing the Processing sketch does is put all the data up until the carriage return (which is represented by `'\n'`) in a “string” variable. A string variable is a fancy programming way to represent a string of characters, which can be a word, a sentence or in this case, a bunch of sensor data. Here’s the code that makes Processing do that:

```
void serialEvent(Serial myPort) {

    String myString = myPort.readStringUntil('\n');
```

So now we have a string variable called `myString` with our sensor data in it. But there are two different sensor data pieces in that string! Remember? Luckily, the values are delimited (a fancy way to say that they are divided by a marker) with a comma so we can easily split them apart. To do this you will need to add some code in the “else” portion inside of the `serialEvent` handler. Put the code where the comment reads “add parsing sensor into array code here.”

```
    else {
        // split the string at the commas
        // and convert the sections into integers:

        //add parsing sensor into array code here
```

Type this line of code:

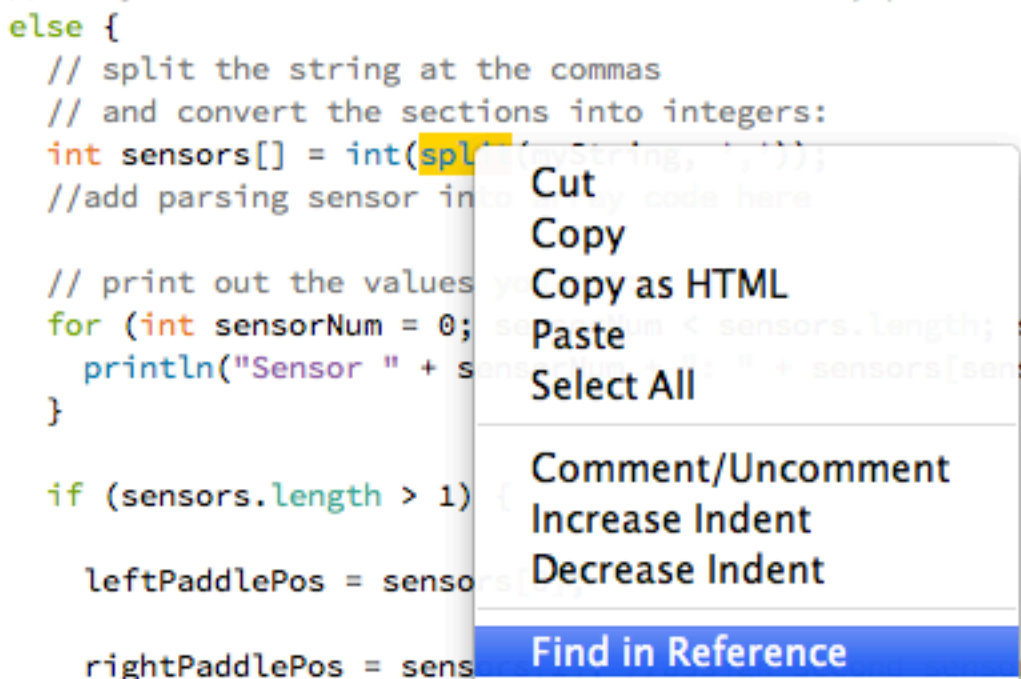
```
int sensors[] = int(split(myString,','));
```

The first part of the code (`int sensors[]`) makes an array of integers called `sensors`. An array is a way to store multiple pieces of information - it’s kind of like a bunch of mailboxes. You put data into each space in the array and then you can access them if you know where you put them. This array holds two integers: the first and second sensor variables that we got over Serial communication. We’ll go over accessing the data in the arrays later.

On the other side of the equal sign is the data we are putting into the array. In this case we are using the “`split`” function to split up the string, called `myString`, each time we see a delimiter. In this case the delimiter is a comma, but really we could put anything inside of the single quotation marks (for example we could use a `!`) and `split` would split the string up into different portions each time it saw the new delimiter (each time it sees a `!`).

The “int” just after the equal sign simply means that Processing should treat each value that gets parsed out (using the `split` function) as an integer.

Feel free to highlight “`split`,” right click on it with your mouse and select “Find in Reference.” This will find the `split` function in the Processing Help File, and will give you more information about how the function works, some example code, and even other functions that are similar to `split`. Any function that shows up in orange text is a “reserved” word, which means that Processing already knows about it and you can look it up in the reference.



If you uncomment (delete the ‘//’) the three lines below the comment that says “//print out the values you got:” you will be able to see the values that you parsed out when you press Processing’s Run button (it’s in the upper left corner and it looks like a play button). The values will show up in the black console area below where you write your code. Leave these lines uncommented and use them to troubleshoot your code and sensors if you run into errors.

- Now scroll down a little farther to find these lines of code:

```
if (sensors.length > 1) {

  leftPaddlePos = sensors[0];
  //assign second sensor array value to rightPaddlePos here

  lpp = (float)leftPaddlePos;

  //reassign rightPaddlePos to the variable rpp of type float
  //println(rpp);

  //adjust these mapping functions to fit the sensors values y
  lpp = map(lpp, 0, 1032, 1, height);
  //rpp = map(rpp, 500, 900, 1, height); //photocell
  //rpp = map(rpp, 200, 1024, 1, height); //soft pot
  //rpp = map(rpp, 150, 275, 1, height); //flex sensor
```

The first of these three lines of code (comment lines don't count as code) take the data from the first place in the sensors array (remember, computers start counting at zero) and assign its value to the variable "leftPaddlePos."

The second line reassigns the value from the integer variable leftPaddlePos to the "float" variable lpp. We need to do this so that the paddle moves smoothly instead of jumping a little each time.

The last uncommented line remaps lpp to a large range. Originally the sensor value of the leftPaddlePos only went up to 1023. But what if the screen on which you want to play Pong is larger (or smaller) than 1023 pixels? That's why we need to remap the value. The "map" function does some simple algebra and stretches the first value in the parenthesis (lpp) from wherever it lies between the next two values in the parenthesis (0 and 255) to where it would lie relationally between the last two values (1 and height). "height" is a reserved word that Processing sees as the height of the window as represented by the number of game pixels.

You should have noticed by now that your paddle slides off the bottom edge of the game window when you crank the potentiometer all the way up. This is because the values in this last line of code are not quite right; you will need to change one of the values in the code, but which one? Remember that your Arduino analog sensor values go up to 1023.

3. Next you need to recreate the three lines of code for the right paddle. I'm not going to help you too much with this one, other than to tell you that the sensor value is stored in the array here: `sensors[1]`, and that the variables `rightPaddlePos` and `rpp` have already been declared for you. If you do it correctly you should be able to control both paddles, each with a different sensor.

If you are having trouble keeping the right paddle on the screen, or it's not moving far enough up and down the screen, you can either uncomment one of the alternative lines for `rpp` or try playing with the values in your own `map` function. You should be able to see the highest and lowest values that your sensor will give you in either the Arduino Serial Monitor or in the Processing console. Just remember to close Arduino before running your Processing sketch; you can't run two different software applications listening to the same Serial communication.

Here is what the portion of my code that parses out Serial data looks like now:

```
// if you have heard from the microcontroller, proceed:
else {
  // split the string at the commas and convert the sections into integer
  int sensors[] = int(split(myString, ',')); //add parsing sensor into a
  // print out the values you got:
  for (int sensorNum = 0; sensorNum < sensors.length; sensorNum++) {
    println("Sensor " + sensorNum + ": " + sensors[sensorNum]);
  }

  if (sensors.length > 1) {
    leftPaddlePos = sensors[0];
    rightPaddlePos = sensors[1]; //assign second sensor array value to ri
    lpp = (float)leftPaddlePos;
    rpp = (float)rightPaddlePos; //reassign rightPaddlePos to the variabl
    //println(rpp);

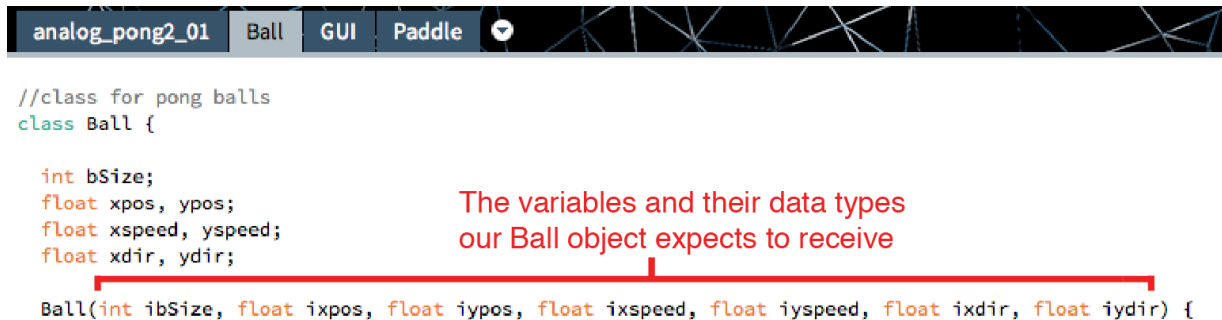
    //adjust these mapping functions to fit the sensors values you're rec
    lpp = map(lpp, 0, 1032, 1, height);
    rpp = map(rpp, 500, 900, 1, height); //photocell
    //rpp = map(rpp, 200, 1024, 1, height); //soft pot
    //rpp = map(rpp, 150, 275, 1, height); //flex sensor
  }
  // when you've parsed the data you have, ask for more:
  myPort.write("A");
}
}
```

4. So you can control the paddles, but there's no ball bouncing the screen yet. Not too much fun, huh? Well, it's time to complete the game with a ball. Go to the bottom of the setup function and look for the comment that reads "//add new instance of ball object called Pong here." Either delete that comment, or create a blank line below it inside of the setup function and type the following code:
 pong = new Ball ();

Here's what my code looks like now that I've added that line:

```
leftPaddle = new Paddle(padWidth, padHeight, distWall, leftPaddlePos); //init right
rightPaddle = new Paddle(padWidth, padHeight, width - distWall, rightPaddlePos); //
pong = new Ball (); //add new instance of Ball object called pong here
}
```

Now we need to figure out what “arguments” we need to pass to the `Ball` object, to figure this out click on the tab labeled `Ball`. Arguments are values that the code in the object header explicitly asks for. This is how you pass information to the objects you create and use. Often these arguments will be pieces of information that vary in some way through out the code.



```
//class for pong balls
class Ball {

  int bSize;
  float xpos, ypos;
  float xspeed, yspeed;
  float xdir, ydir;

  Ball(int ibSize, float ixpos, float iypos, float ixspeed, float iyspeed, float idxir, float iydin) {
```

The variables and their data types
 our Ball object expects to receive

The arguments are listed inside the parenthesis to the right of the word `Ball`, right at the bottom of the image above. They are listed first by variable type and then by name. The values are delimited with commas. There are a total of seven arguments that this `Ball` object needs to be passed. Below is a brief list of what each argument means and what values you should pass to the `Pong` instance. (Remember, that’s back in the `analog_pong2_01` tab. Also, if you’ve been doing this right your tab should be called `analog_pong2LRC_03` or something like that, because you’ve been saving out versions of your code.) We will go into some of these variables in depth later.

- `ibSize`: This is the initial size of the ball in pixels, set it to 15
- `ixpos`: This is the initial x position of the ball, set it to `width/2`
- `iypos`: This is the initial y position of the ball, set it to `height/2`
- `ixspeed`: This is the initial speed of the ball on the x axis, set it to 8
- `iyspeed`: This is the initial speed of the ball on the y axis, set it to 2
- `idxir`: This is the initial direction of the ball on the x axis, set it to 1
- `iydir`: This is the initial direction of the ball on the y axis, set it to 1

Here’s what that last line in your `setup` function should look like if you’ve done it right:

```
pong = new Ball(15, width/2, height/2, 8, 2, 1, 1);
```

You should run the Processing sketch now to see how the game plays with these values. After you’ve played your game for a while close the game window and start changing the arguments you pass to the `Ball` object. Each time you change a number, run the Processing sketch again to see how it changes the play of the game.

What does this tell you about how Processing uses x and y to draw pixels in the window? Where does it start graphing? In the middle of the game screen, or at one of the corners? If you change the second argument number to zero, where does the ball appear when you start the game? What about the third number?

How do the fourth, fifth, sixth and seventh argument numbers correlate to the angle of the ball's trajectory when you play the game?

If I want the ball to start out traveling toward the left hand player what argument do I have to change? What if I want it to be aimed more at the top of the window than the bottom, what argument do I change then?

Try using "Find in Reference" to figure out how to use the `random` function and then use the `random` function at least twice to make the ball head at a random angle when you start the game.

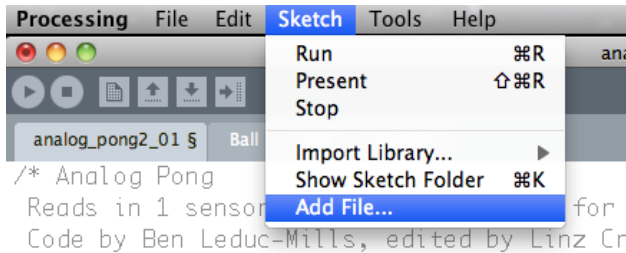
Once you've figured out how these arguments work change them back to the initial values and continue on to the next steps to make the game better.

5. That's cool and all, but it only works the first time you start your game right? Look inside the `Ball` object code to find where it resets `xpos`, `ypos`, `xdir` and `ydir`. There are three places these variables are reset. One is for after player one scores, one is for after player two scores, and the last is for when the game resets. Replace these twelve numbers (four variables, three different places) with numbers (or random functions) of your choosing.
6. This next step is really, really easy in comparison to what you just did. Now we're going to change the size of the game window. To do this simply change the numbers of the `size` function, which can be found inside the `setup` function. Currently the width is set to 1000 and the height is set to 600.

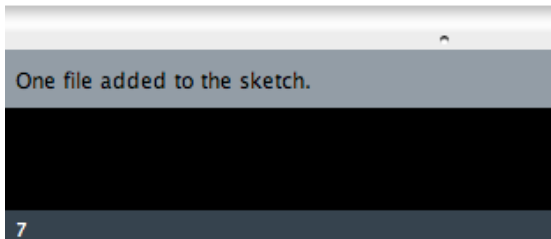
```
void setup() {
  size(1000, 600); //Size of Game Window
  rectMode(CENTER);
}
```

These numbers represent the width (x) and height (y) of the game window in pixels. Try to make the game window the same proportions as a tennis court by changing the numbers 1000 and 600.

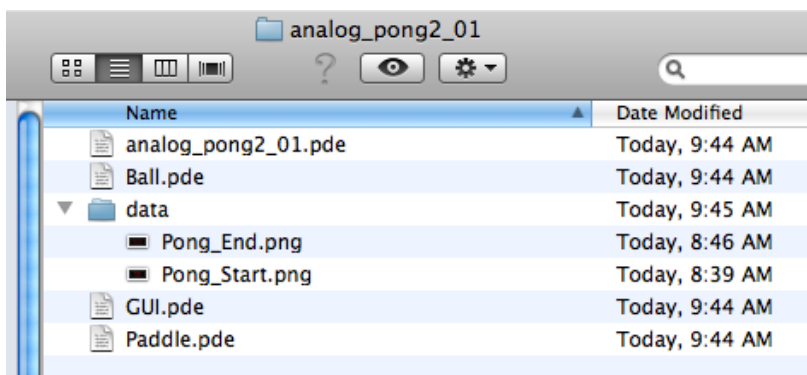
7. Ready for something a little tougher? Next we're going to add some Start Game and Game Over screens. I've included two images to use called Pong_Start and Pong_End, but feel free to make better images. Programming these screens is going to take a bunch of different steps. The first step is importing the image files. To do this select Add File from the Sketch menu and find the files you wish to add to your Sketch.



Processing plays well with the following image file types: .jpg, .png and .gif. Processing also likes raster and vector images (these last two are math-based image types instead of pixel image types, meaning they will not distort when scaled). If everything goes well you should see a message that reads "One file added to the sketch" below the area where you write code.



This means Processing has automatically created a folder called Data inside your Sketch folder with the image file inside of the Data folder. You can also drag and drop images and fonts (you will also need to import fonts to use them) into this folder as an alternative way to complete this first step of adding files. After you have completed this step for both images check your Sketch folder and make sure Processing has created a data folder with your files in it.



8. Next you will create an object to store the first Pong_Start image. You will create this object to inside the `setup` function. You need to create the object so Processing has a place to store the image. To create your image object, type this line above your `setup` function:

```
PImage pStart;
```

An object is like a variable but it contains more information and functions, and it's for a Class, don't worry about Classes right now, you'll get into classes later if you pursue programming. The object name can be anything you like (I named mine `pStart`) as long as it makes sense to you. But the Class, `PImage`, needs to be the same whenever you are using an imported image. Make sure you type this line above the `setup` function because you want to be able to use this variable anywhere in your sketch. If you wrote it inside the `setup` function or the `draw` function you would only be able to use it inside those functions.

Go back and do this step again for the Pong_End image. Remember to name this second object something different (I named mine `pEnd`).

Now you have two instances of objects from the `PImage` Class (`PImage` stands for Processing Image) in which you can store the information that are your actual images. Next, you have to assign the data files to the objects you created. To do this you will assign the image data file to the `pStart` and `pEnd` objects (or whatever you named it if you didn't use `pStart` and `pEnd`) using the `loadImage` function as typed below. Make sure to include quotation marks around the image data file names as well as the file extensions (`.jpg`, `.png`, or `.gif`). The file names are also case sensitive, so make sure you type them exactly as they appear in the folder. Type this code in the `setup` function.

```
pStart = loadImage ("Pong_Start.png");
pEnd = loadImage ("Pong_End.png");
```

Here's what the lines I added to my code look like:

```
boolean firstContact = false; // Whether we've heard from the microcontroller

PImage pStart; //PImage object for Start Screen
PImage pEnd; //PImage object for End Screen

void setup() {
  pStart = loadImage("Pong_Start.png"); //Assigning the data or file
  pEnd = loadImage("Pong_End.png"); //Assigning the data or file

  size(1000, 600); //Size of Game Window
```

9. Sweet, you've got the image files imported and you're almost ready to place them in the game. But first we need to write a little code so the game knows to pause while displaying the screens and wait for the player to press a button before continuing.

To do this we need to create a variable, an `if` statement, and add a little code to the `keyPressed` handler (the `keyPressed` handler can be found at the bottom of the main `analog_pong2` tab).

One thing at a time though. First, create a boolean variable named `started` (or whatever makes sense to you) and assign it a value of `false` (or zero) just above the `setup` function. Here's what that line of code looks like:

```
boolean started = false;
```

Next we need to add an `if` statement at the very end of the draw loop that displays the Start screen as long as the value of the variable `started` is `false` (also represented as zero). Make sure you put the `if` statement at the end of the draw loop because otherwise Processing will draw the image but then draw game graphics over it. Here's what that empty `if` statement looks like (we'll put more code inside the `if` statement in a second); make sure you use two equal signs and not just one:

```
if (started == false) {  
  
}
```

Okay, you're almost ready to actually draw the image in your Processing Window. To do this you just need to type one more line using the `image` function. The `image` function has three parameters or variables: the image object you just created, `x` position and `y` position. Type the following line, and substitute the `x` and `y` positions where you wish your image to display for the variables `x` and `y`. Remember that these variables indicate where the upper left corner of your image will start. Place this line of code inside the `if` statement you just created:

```
image (pStart, x, y);
```

To control the size of the image simply add two more variables after `x` and `y` for the width and height of the image. If you leave width and height out of the function, or just give them a value of 0, the image will draw itself at the size of the original picture in the data file you imported.

You can use the `width` and `height` variables instead of numbers, and the screen will resize itself depending on the numbers you put in the `size` function. The variables `width` and `height` are global variables that are always equal to the width and height of your canvas window. Don't make Processing draw the image larger than the original file unless you are okay with image distortion (or you are using a vector file format like `.svg`).

Here's what the code we just added looks like in my sketch at this point:

```
void draw() {
  background(50);
  showGUI(); //shows scores, etc. (see GUI tab)
  pong.display();
  pong.update();
  leftPaddle.display(lpp); //show left paddle
  rightPaddle.display(rpp); //show right paddle
  if (started == false) {
    image(pStart, 0, 0, width, height); //start
    println("Start");
  }
}
```

We are almost done with the Start screen. The last thing we need is some code inside of the `keyPressed` handler that sets `started` equal to `true` (or 1) if a button is pressed while `started` is equal to `false`. We also need to make sure that the other code inside the `keyPressed` handler only happens if the variable `started` is `true`. Here is the `keyPressed` handler with those two `if` statements added to it:

```
void keyPressed() {
  if (started == true) { //added for Start Screen
    pong.keyPressed();
  } //added for Start Screen
  if (started == false) { //added for Start Screen
    started = true; //added for Start Screen
  } //added for Start Screen
}
```

The `keyPressed` handler is a function that is called anytime the user presses a key on the keyboard and it can be found at the bottom of the main `analog_pong2` sketch. It is similar to the `serialEvent` handler because it is a piece of code that will interrupt almost everything else when the event it is designed to handle occurs. `while` loops are an exception to this rule, because the computer never reaches the end of the draw loop where the `serialEvent` and other event handlers execute. If we used a `while` loop instead of an `if` statement to display the Start screen, Processing would stay stuck inside the `while` loop because it would never update the variable `started`. The `pong.keyPressed` function can be found inside the `Ball` class.

10. That's the Start screen, now let's do the End screen. This will be a lot easier, all you have to do is find the `showGUI` function in the GUI tab and enter two lines of code. Here's the code we will be changing:

```
void showGUI() {

  stroke(150);
  line(width/2, 0, width/2, height);

  textSize(16);
  text("Player 1: " + playerOne, 100, 20); // player 1 score display
  text("Player 2: " + playerTwo, 600, 20); // play 2 score display

  if (oneWins == true) { //if player one wins
    textSize(24);
    text("Player 1 WINS!!!", width/2 - 95, height/2 - 50);
    text("Press 'R' for New Game", width/2 - 135, height/2 - 20);
  }

  if (twoWins == true) { //if player two wins
    textSize(24);
    text("Player 2 WINS!!!", width/2 - 95, height/2 - 50);
    text("Press 'R' for New Game", width/2 - 135, height/2 - 20);
  }
}
```

Can you guess where we need to add the line of code that displays the image of our End screen? Remember that we want to be able to see the text that says "Player 2 WINS!!!" but we don't really care about the text that says "Press 'R' for New Game" (because it already says the same thing on the End Game screen).

```
void showGUI() {

  stroke(150);
  line(width/2, 0, width/2, height);

  textSize(16);
  text("Player 1: " + playerOne, 100, 20); // player 1 score display
  text("Player 2: " + playerTwo, 600, 20); // play 2 score display

  if (oneWins == true) { //if player one wins
    textSize(24);
    image(pEnd, 0, 0, width, height); //display End Game Screen
    text("Player 1 WINS!!!", width/2 - 95, height/2 - 50);
    //text("Press 'R' for New Game", width/2 - 135, height/2 - 20);
  }

  if (twoWins == true) { //if player two wins
    textSize(24);
    image(pEnd, 0, 0, width, height); //display End Game Screen
    text("Player 2 WINS!!!", width/2 - 95, height/2 - 50);
    //text("Press 'R' for New Game", width/2 - 135, height/2 - 20);
  }
}
```

Display the End Game image before Processing writes the text about which player won so that the text gets written over top of the image. We don't want to see the text about pressing 'R' for a new game, so simply comment it out. You will need to change the x and y positioning of the text declaring the winner so that it's not in the middle of everything. Go ahead and do that before moving on to the next step.

11. You may have noticed while playing this version of Pong that the angle of the ball doesn't change a whole lot. In real Pong, the angle of the ball bounce is affected by where the ball hits the paddle. This aspect makes the game much more interesting; let's change the hit test code so that the ball bounces differently off the paddle. Like most aspects of game coding there are many different ways to do this. The first step is to think about what we want to happen. If you don't know exactly what you want to have happen it is difficult to define it with code. We know we want the angle of ball's trajectory to change depending on where it hits the paddle, but how exactly do we want it to change? I've outlined a couple options in the following pages, but you should feel free to pursue your own options.

- A. If the ball hits the middle of the paddle it does not change its angle. If the ball hits the upper portion of the paddle its angle is changed so it is headed more towards the top of the screen, and if the ball hits the lower portion of the paddle its angle is changed so it is headed more towards the bottom of the screen.
- B. Another option that is very similar (but a little harder) to the above option is that the ball bounces as described above, but the ball's angle change is greater the farther away from the center of the paddle the ball hits. This option still needs a center area of the paddle that does not affect the ball's trajectory.
- C. A third option is that it is not actually the point on the paddle that affects the ball's bounce, but the speed and direction the paddle is moving when it makes contact with the ball. This approach is probably the hardest of the three outlined, as well as the most realistic. In order to code this option you would need to use two variables instead of one: the direction the paddle is moving and the speed at which the paddle is moving. To do this you would need to remember the paddle's last position and compare it to the paddle's current position. Either that or you could keep track of the player's last sensor reading and compare it to the most current sensor reading.

I'm going to pursue the second of these three options and take you through the process of writing pseudo-code as well as the actual code, so you can get a feel for the problem solving process involved in coding.

First off, I know exactly how I want my ball and paddle to interact, but I'm not sure where I need to put the code or how much I need the angle to change. We'll worry about figuring out the code first (using place holder numbers to affect the ball's bounce), then we'll worry about where the code goes. Only after we've got something that works the way we want will we worry about changing the values that affect the angle of the ball's bounce. I know it sounds complicated, but stick with me.

Let's write out what we want to have happen in pseudo-code first. For those of you not familiar with pseudo-code, it is simply a way to represent code as a mix of code and normal writing so that humans have an easier time thinking about what we are trying to make the computer do. Here's how I would think about our paddle and ball problem in pseudo-code:

```
//start pseudo-code

find the total speed of the ball by calculating using
the x and y variables;

if ( the ball hits the right paddle ) {
figure out where the ball hit the paddle and change
the ball's angle using the total speed variable;
make the ball bounce off the paddle;
}

if ( the ball hits the left paddle ) {
figure out where the ball hit the paddle and change
the ball's angle using the total speed variable;
make the ball bounce off the paddle;
}

//end pseudo-code
```

There are two things I'd like to clear up at this point. First, yes, we're going to delete (or at least comment out) the current hit test. I'm doing this simply because it's easier to break up the code for beginners. If you can write an addition to the existing hit test that makes it function like the pseudo-code outlined above, go for it. Just make sure you comment your code, because I can almost guarantee it's going to get confusing.

Also – if you look at the code we are already working with you will notice that the code does not use an angle to decide how the ball is bouncing, it uses a Cartesian graphing system. (This is just a fancy way to say that it uses an X and Y coordinate-based system.) So we will need to be able to either use the existing X and Y system, or convert the X and Y system to an angle value, change the ball's trajectory angle and then convert that new angle back to the original X and Y system to change the ball's trajectory. This is the reason for the first line of the pseudo-code I outlined. We need to do some math to figure out the ball's trajectory angle and speed (when these are combined it is known as a vector) before we can change the X and Y speed values in the code.

12. Next up let's write the actual code represented by the pseudo-code. As with most code there are a couple different ways to do this and mine only represents one of those ways. Feel free to write your code another way. The cool thing is that there is no right answer; your code either works or it doesn't! (And if the code doesn't work that doesn't mean that it's wrong, it might just mean that you have to do some more work on it.)

Our first move will be to delete or comment out the existing hit test. Luckily this will keep us from scratching our heads too much when we are trying to figure out where the new code needs to be inserted. See if you can find these first seven (nine with comments) lines of code inside of the update function (it's in the Ball tab). Comment them out like this:

```
void update () {

    xpos = xpos + (xspeed * xdir);
    ypos = ypos + (yspeed * ydir);

    /*
        if ( (xpos > rightPaddle.xPos - bSize &&
            xpos < width - bSize &&
            ypos < rpp + rightPaddle.pHeight/2 - bSize &&
            ypos > rpp - rightPaddle.pHeight/2 - bSize) ||
            //above is checking right paddle
            //below is checking left paddle
            (xpos < leftPaddle.xPos + bSize && xpos < 0 +
            bSize &&
            ypos < lpp + leftPaddle.pHeight/2 - bSize &&
            ypos > lpp - leftPaddle.pHeight/2 - bSize) )

        {
            xdir *= -1; //change ball direction
        }
    */
}
```

What I did in my code was create a total of five different functions that we will use to control the hit test and how the ball bounces after the hit test. You may not have created functions before, but it's actually pretty easy.

Scroll down to the bottom of the Ball tab and write the following lines of code. These lines are called function “headers.” These function headers are just the beginning, and we will fill them in later. But before we do that I will explain exactly what functions are and what these specific functions will be doing before we continue.

```
void newGame() {  
  
    playerOne = 0;  
    playerTwo = 0;  
    oneWins = false;  
    twoWins = false;  
    reset();  
}  
  
//function headers start here  
  
void calcRadius() {      //added to calculate radius/vector of t  
}  
  
void rPadHitTest() {      //added to hit test for right paddle  
}  
  
void lPadHitTest() {      //added to hit test for left paddle  
}  
  
void rPadAngle() {        //added to change the ball's angle wher  
}  
  
void lPadAngle() {        //added to change the ball's angle wher  
}  
}
```

So let’s talk about functions and function headers before we talk about what each one of these functions will actually do in our code.

A function is a way to keep your code separated into different pieces. It also allows you to use the same pieces of code over and over again at different points without having to retype them. Often a programmer will even use the same piece of code on different pieces of data. Ideally a function will complete one task and one task only. Usually it has a descriptive name that makes it easier to figure out exactly what your code is doing when the function is called.

In Processing the function header is made up of four parts. The first part is the data type that a function may send back to the main portion of code. Here is an example of a function header, for a function named `myFunction`, that returns an `integer`:

```
int myFunction () {  
  
}
```

(Don't worry about this code, don't add `myFunction` to anything, this is just an example.)

But what does “void” mean? All the functions we are creating right now start with `void`, right? `void` simply means that the function will not return any values. This is why the functions we are creating all start with `void`, because they don't send or return any variable values. These functions will reassign certain variable's values, but they won't actually return any values.

The second part of the function is the name of the function. Just like a variable, this name can be anything as long as it contains no spaces. You can name it `spatula`, `stopTime` or even `whatADumbName4AFunction`. Just make sure that the name describes what the function does so that people know how to use it. The example function above is named `myFunction`.

The third part of a function is the parenthesis after the name and anything inside the parenthesis. So far all our functions have had nothing inside the parenthesis, but if the functions were expecting Processing to send, or “pass” them some information, they would have the variable type and the name of the variable (as it is used inside the function, not necessarily what the variable that is being “passed” is named) inside these parenthesis. You could even “pass” a function multiple variables. This would make it necessary to list both variable types along with the variable names as they are used inside the function, only with a comma between the two so Processing can tell them apart. Here's an example of the `myFunction` header that is expecting you to send it two integers:

```
int myFunction (int variable1, int variable2) {  
  
}
```

I could have sent it a Boolean and an integer or any other combination of data types. You already “passed” the Ball function multiple variables in step #4, as well as a couple other places. This shouldn’t come as a complete surprise to you.

The fourth part of a function header is the curly brackets that contain all the function code that comes after the header. You’ve seen these a lot and you’ve probably become familiar with how much of a pain they can be. The important thing to understand is that when you see one curly bracket going this way { you will always need another curly bracket facing the other way: }. Think of the curly brackets like the bread in a sandwich; whenever you have one piece of bread on one side of the sandwich you will also need a piece of bread on the other side of the sandwich. Just make sure you eat your code sandwich one byte at a time. I’m sorry, I promise that’s the only bad joke in this activity.

Here’s a list of the five functions we are creating and what they will do:

calcRadius will calculate the vector of the ball’s trajectory for use in later functions. I named the function calcRadius because the vector is also referred to as the radius.

rPadHitTest tests to see if the ball hit the right paddle using the X and Y coordinates of both the ball and the paddle. If the ball and paddle hit, this function will call the rPadAngle function.

IPadHitTest tests to see if the ball hit the left paddle using the X and Y coordinates of both the ball and the paddle. If the ball and paddle hit, this function will call the IPadAngle function.

rPadAngle changes the xspeed and yspeed of the ball, depending on where the ball hit the right paddle, while keeping the radius, or vector, of the ball constant (so it doesn’t speed up or slow down in the process).

IPadAngle changes the xspeed and yspeed of the ball, depending on where the ball hit the left paddle, while keeping the radius, or vector, of the ball constant (so it doesn’t speed up or slow down in the process).

13. Ok. Now we've got to do some math. No... wait, please don't leave! In order to make any video game that's any fun at all, sooner or later you're going to have to do some math. In fact, you've been doing math all along, we just never told you before now. But know we're going to do a little trigonometry, so you may want to brush up on trig (at least understand Sine, Cosine and the difference between radians and degrees), or figure out a different way to write this code if you really don't like trigonometry.

Here's the problem: Right now we have four different variables that control the direction (or angle) of the ball's trajectory. Those variables are `xdir`, `ydir`, `xspeed` and `yspeed`. We don't care about `xdir` and `ydir` since the code already changes these for us. But, if we just change `yspeed`, depending on where the ball hits the paddle we will wind up with a change in the ball's overall speed. You can do math so that `yspeed` doesn't pick up too much, but that also means that the ball's angle won't change all that much either. In order to make sure that doesn't happen, we need to change both `xspeed` and `yspeed` so the overall angle changes the way we want, but the overall speed doesn't. That way would involve a little guesstimation, so we might as well do the math to convert the resulting vector of `xspeed` and `yspeed` to radians, change it a little in the direction we want and then convert the resulting radian value back to the `xspeed` and `yspeed` values. Are you still with me? Dang, I guess that means I actually have to do the math, huh?

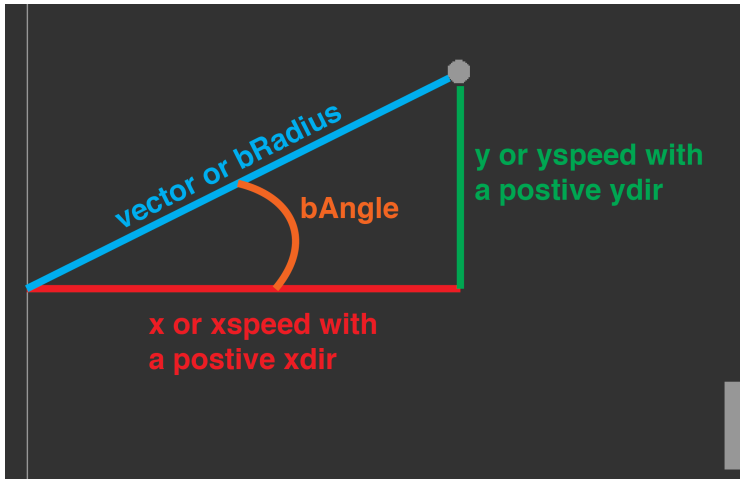
Here is what we know (this is basic trigonometry, which despite what you may think, is actually very useful in a whole bunch of different situations):

```
float x = cos(radians(angle)) * radius;
```

We also know:

```
float y = sin(radians(angle)) * radius;
```

We're solving for the angle and we know everything else in the equation, so the trick is to isolate the angle. It doesn't matter whether it's displayed in radians or degrees.



First I figured out the value of radius, which we need to keep constant so the ball doesn't move any faster than it did before. This radius variable actually represents the speed that the ball travels. In fact the radius is a vector if you want to get into even more math. To figure out the radius value I used the Pythagorean theorem; $a^2 + b^2 = c^2$, where c is the value (radius) we are solving for. With my initial values of 8 (for x) and 2 (for y) my math looks like this:

$$8^2 + 2^2 = c^2 \quad \text{or} \quad 64 + 4 = c^2 \quad \text{or} \quad 68 = c^2 \quad \text{or} \quad \sqrt{68} = c \quad \text{or} \quad 8.25 = c$$

Remember though, your values may be different or may change throughout the game. (In fact we're going to make them change later on.) So you should create a new variable called something like "bRadius" and solve for your particular radius inside the Pong code. Make sure that you declare the "bRadius" variable inside of your Ball object with a line like this:

```
class Ball {
float bRadius; //add this line to create your variable
```

The function you would use to solve for a square root in Processing is `sqrt()`. Remember, you can easily use "reference" to figure out exactly how to use the square root function. I'm not going to help you create the "bRadius" variable but I will show you what your function should look like now:

```
void calcRadius() { //added to calculate radius/vector of bal
    bRadius = sqrt(pow(abs(xspeed), 2) + pow(abs(yspeed), 2));
}
```

The `pow` function is used to raise a number to a power. The `abs ()` function is used to find the absolute value of a number. The absolute value of a number simply means that if the number is a negative number it is turned into a positive number and if it is a positive number it stays a positive number. By now you should feel comfortable using reference to figure out how these functions work if you have additional questions.

Now we have all the variables we need in order to solve for the angle of our equation. I'm going to go through this process with the X value in order to find our initial angle value, but I could just as easily use the Y value and the slightly different equation to find the same angle. Anyway, here's how I juggled the equation we started with in order to solve for the angle value.

$$\text{acos}(8/b\text{Radius}) = \text{radians (angle)}$$

This equation is in pseudo-code at this point because, although I am using the `acos` function to reverse the `cos` function (cosine), there are no variable types or semicolons, and I'm using a single equal sign instead of two. Also be aware that the answer to this equation will be in radians. Radians are simply a different way to represent degrees, with values from zero to 2π (or about 2.68) instead of zero to 360. Here's my answer in radians:

$$\text{acos}(8/b\text{Radius}) = 0.24680881$$

or if you prefer degrees:

$$\text{degrees} = 0.24680881 / 2\pi * 360 \quad \text{or} \quad 0.24680881 / 2\pi * 360 = 140^\circ$$

By now you may have forgotten exactly what it is we are trying to do. Remember that this angle we just solved for is the current angle of the ball's trajectory before it hits a paddle. Once we have calculate this variable we can use it's value later on in our other functions.

14. Next let's work on those hit tests. Luckily we can work on creating the right paddle hit test first, then copy and paste it and change a couple things so it works for the left paddle as well.

First we need a line of code (called a "condition" because it checks to see if the variables meet the condition) that will check the x position of the ball against the x position of the right paddle. We need to make sure that the x position of the ball is greater than the left edge of the right paddle. Here's what that line of code looks like:

```
xpos > rightPaddle.xPos - rightPaddle.pWidth/2 - bSize/2
```

Then we need to check to see if the ball is above the bottom of the paddle.

```
ypos < rpp + rightPaddle.pHeight/2 + bSize/2
```

Lastly we will need to check to see if the ball is below the top of the paddle. I'm going to let you try to figure this one out on your own. (Hint, it's a lot like the last line of code I gave you.)

Put all three of these lines of code inside the conditional (the parenthesis) of an `if` statement. Use an "and" operator to link these three conditions. In Processing (and most programming languages) the "and" symbol is `&&`. This means Processing requires all three of the conditions to be true in order to allow the code inside each `if` statement's curly brackets to execute. This way we know that the ball is somewhere in the same X and Y area as the paddle, and we are ready to change the ball's trajectory angle and make it bounce.

We'll deal with changing the actual angle in a second, for now call the function `rPadAngle` like this (Make sure you're doing this inside of the `rPadHitTest()` function.):

```
rPadAngle ();
```

Later on inside the `rPadAngle()` function we will do some more math and change the actual angle of the ball.

The very last thing we have to do is change the `xdir` value of the ball so that it actually changes direction as if it were bouncing. That's pretty easy compared to the math we've been doing. All you have to do is assign `xdir` equal to the value of `xdir` multiplied by negative one. That may sound a little tricky. There are two different lines of code you could use to accomplish this:

```
xdir = xdir * -1;
```

or

```
xdir *= -1;
```

All right, this was a long step, but what did you expect? Hit tests are some of the more difficult aspects of game coding. Here's what my `rPadHitTest()` looks like now:

```
void rPadHitTest() { //added to hit test for right paddle
  if (xpos > rightPaddle.xPos - rightPaddle.pWidth/2 + bSize/2 //ch
    && ypos < rpp + rightPaddle.pHeight/2 + bSize/2           //check
    && ypos > rpp - rightPaddle.pHeight/2 - bSize/2)           //check
  {
    rPadAngle();                                               //change
    xdir *= -1;                                                //turn t
  }
}
```

I'm not going to help you figure out how to change this code so that it works for the left paddle, but if you run into trouble try drawing a diagram of how this hit test works and comparing that to how you want your left hit test to work.

15. Now that we are almost done with our new hit test functions we can put them into our `update()` function. In order to do this simply write the following three lines of code inside of `update()`, below the code you previously commented out.

```
void update() {

    xpos = xpos + (xspeed * xdir);
    ypos = ypos + (yspeed * ydir);

    /* commented out for better hit test
    //if hits paddle, change direction
    if ((xpos > rightPaddle.xPos - bSize && xpos < width - bSize
    ypos < rpp + rightPaddle.pHeight/2 - bSize &&
    ypos > rpp - rightPaddle.pHeight/2 - bSize) ||
    (xpos < leftPaddle.xPos + bSize && xpos < 0 + bSize &&
    ypos < lpp + leftPaddle.pHeight/2 - bSize &&
    ypos > lpp - leftPaddle.pHeight/2 - bSize) )

    {
        xdir *= -1; //change ball direction
    }
    */

    //added for new hit test
    calcRadius(); //called to find out current ball vector value

    rPadHitTest();//here is the right paddle hit test

    lPadHitTest();//here is the left paddle hit test
```

16. We are so close to having a hit test that changes the angle of the ball's trajectory. If you want to play the game now, go ahead. It should function exactly like it did before we changed the hit test. Just make sure you save out a different version first in case you messed any of the code up in the process!

You back? Great. We're going to change the angle of the ball's trajectory now by writing some code in our `rPadAngle()` function. Big surprise! The first thing we need to do is declare a new variable that we will use to store and change the ball's angle value. The variable type should be a `float`, because we're going to run into decimals doing math so an integer type variable just won't work. I called my variable `bAngle` (short for ball angle) but as you know by now you can call your variable whatever you want. I declared the variable inside of the `Ball` class because I'm only going to be using it in the `Ball` code.

```
//class for pong balls
class Ball {

    float bAngle = 0; //added to keep track of ball's current angle
    float bRadius; //add this line to create a vector for the ball

    int bSize;
    float xpos, ypos;
    float xspeed, yspeed;
    float xdir, ydir;

    Ball(int ibSize, float ixpos, float iypos, float ixspeed, float iys

        bSize = ibSize;
        xpos = ixpos;
        ypos = iypos;
        xspeed = ixspeed;
        yspeed = iyspeed;
        xdir = ixdir;
        ydir = iydir;
    }
}
```

Before we can use this variable we need to make a couple `if` statements that check to see where on the paddle the ball is hitting. I want to leave a 20 pixel space in the middle of the paddle where the ball's trajectory is not affected, but I do want the upper and lower portions of the paddle to change the ball's trajectory. Before we worry about changing the angle of the ball we need to use the two `if` statements to decide if the angle will change so the ball is headed more towards the top or the bottom of the game window. Here's what that looks like:

```
void rPadAngle() {           //added to change the ball's angle when it
  if (ypos < rpp - 10) {     //checks if y position is above the midc

  }
  if (ypos < rpp + 10) {     //checks if y position is below the midc

  }

}
```

Your two `if` statements inside the `lPadAngle()` function should be almost the same, you just need to change one variable.

Now we are ready to change the actual `bAngle` variable that we use to keep track of the ball's trajectory. In order to do that we will need to use the `map` function to make the ball's distance from the middle of the paddle (except it's not quite the center, because we want a 10 pixel "dead" space in the middle of the paddle) correspond to a range of degrees that we want the ball's angle to change by. This sounds a little complicated so I'll break it down for you. Here's the entire line of code we'll put in the first `if` statement:

```
bAngle = bAngle - (map(abs(rpp - 10 - ypos), 0, 50, 0, 15));
```

The `map` portion of this code takes the difference in distance between the middle of the paddle (`rpp`) with an offset of 10 pixels (minus 10) and the ball's y position (`ypos`) and first turns it into a positive number (this isn't strictly necessary, but it is if you want to copy, paste and change this code) using the `abs` function. Then we simply `map` that value from the range of zero to 50 down to zero to 15. This is because we want the angle to change by a maximum of 15° when it hits the very edge of the paddle. If you want a larger angle change when the ball hits the side of the paddle, you simply have to change the last number in this `map()` function.

17. Now we're ready to plug our equations that change `xspeed` and `yspeed` into the code. These are the pieces that actually change how the ball functions, the rest was legwork so we could get to these pieces of code at the right time with the right variable values. In order to do that we've got to go back to our trigonometry skills using Cosine and Sine. Add the following lines of code in the `rPadAngle` function, below the `map` function you just added:

```
xspeed = cos (radians(bAngle)) * bRadius;  
yspeed = sin (radians(bAngle)) * bRadius;
```

Luckily you can tweak the code a little, save your code, run the Processing sketch, see how it works and then continue to work on the code if you need to. The key concept is to just change a little bit of the code at a time so you can tell how it changed the game play of your Processing sketch.

18. Finally, copy and paste the code from inside the first `if` statement's parenthesis to inside the second `if` statements parenthesis. You will need to change the code a little to make it work with the lower section of the paddle instead of the upper. Make sure you change the code that checks the ball's distance from the center of the paddle, and the code that changes the angle. Save your code (you've been saving different versions of your code this whole time, right?) and press the Run button to see if everything works. Remember, this will only change how your right paddle operates. Testing is the best part of game coding; you have to play your game a bunch to make sure nothing is broken. If something is broken, note exactly how the game play is broken so you have a better idea of where to look in your code. If you need help there is an image of my code below. Make sure your code works exactly how you want it to before moving on to the next step.

```
void rPadAngle() {          //added to change the ball's angle when
  if (ypos < rpp - 10) {    //checks if y position is above the mid
    println(bAngle);
    bAngle = bAngle - (map(abs(rpp - 10 - ypos), 0, 50, 0, 15));
    xspeed = cos (radians(bAngle)) * bRadius; //change ball's x s
    yspeed = sin (radians(bAngle)) * bRadius; //change ball's y s
    println(bAngle);
  }
  if (ypos < rpp + 10) {    //checks if y position is below the mid
    println(bAngle);
    bAngle = bAngle + (map(abs(rpp + 10 - ypos), 0, 50, 0, 15));
    xspeed = cos (radians(bAngle)) * bRadius; //change ball's x s
    yspeed = sin (radians(bAngle)) * bRadius; //change ball's y s
    println(bAngle);
  }
}
```

I put `println(bAngle)` commands in my code so I could see the amount that this code changes `bAngle` every time the ball hits the right paddle. Feel free to put those commands in or leave them out.

19. Go ahead and copy and paste the code from inside the `rPadAngle()` function to the `lPadAngle()`. In order to repurpose this code from right to left you will need to change a total of six things in the `lPadAngle()` code after you copy and paste it in order to make it work properly with the left paddle. I felt bad for you so I highlighted the changes made between the two functions in the code below:

```
void lPadAngle() {
    if (ypos < lpp - 10) { //checks if y position is above the midc
        println(bAngle);
        bAngle = bAngle + map(abs(lpp - 10 - ypos), 0, 50, 0, 15));
        xspeed = cos(radians(bAngle)) * bRadius; //change ball's x sp
        yspeed = sin(radians(bAngle)) * bRadius; //change ball's y sp
        println(bAngle);
    }
    if (ypos < lpp + 10) { //checks if y position is below the midc
        println(bAngle);
        bAngle = bAngle - map(abs(lpp + 10 - ypos), 0, 50, 0, 15));
        xspeed = cos(radians(bAngle)) * bRadius; //change ball's x sp
        yspeed = sin(radians(bAngle)) * bRadius; //change ball's y sp
        println(bAngle);
    }
    // .....
}
}
```

20. By now you may have noticed that there is a slight issue with the paddle code. Sometimes when the ball hits the very edge of the paddle it gets kind of stuck inside the paddle. That's no good. Sometimes you write code that almost works how you want, but there is an issue that you need to fix like this. In this case we can simply reset the x position of the ball to the front of the paddle after they hit. To do that just add the following line of code inside the last two curly brackets of the `rPadAngle` and `lPadAngle` functions:

```
xpos = rightPaddle.xpos + rightPaddle.pWidth/2 + bSize/2;
```

Here's what the code looks like in with my left paddle functions:

```
void lPadAngle() {
    if (ypos < lpp - 10) { //checks if y position is above the middle
        println(bAngle);
        bAngle = bAngle + (map(abs(lpp - 10 - ypos), 0, 50, 0, 15)); //
        xspeed = cos (radians(bAngle)) * bRadius; //change ball's x speed
        yspeed = sin (radians(bAngle)) * bRadius; //change ball's y speed
        println(bAngle);
    }
    if (ypos < lpp + 10) { //checks if y position is below the middle
        println(bAngle);
        bAngle = bAngle - (map(abs(lpp + 10 - ypos), 0, 50, 0, 15)); //
        xspeed = cos (radians(bAngle)) * bRadius; //change ball's x speed
        yspeed = sin (radians(bAngle)) * bRadius; //change ball's y speed
        println(bAngle);
    }
    xpos = leftPaddle.xPos + leftPaddle.pWidth/2 + bSize/2; //correct
}
}
```

You will need to change four parts of this code to make it work with the left paddle. I'm going to let you try and figure out those changes on your own.

21. This next addition should be pretty easy in comparison to changing the bounce of the ball off the paddles. We're going to make a counter that keeps track of time passed in the game. Then we're going to use that counter to speed up the game as time passes. Counters are incredibly useful in all kinds of programming; whether it's a game, a robot or a piece of social media, it's probably got at least one counter in the code.

Like most of the additions we've been making to our Pong game, this one needs a variable. You can call it whatever you want, just make sure the variable is an integer equal to zero that gets declared before `setup` in the main section of code.

```
boolean started = false; //boolean to keep track of whether we have st

int timePlayed = 0; //added to create timer

void setup() {
  pStart = loadImage("Pong_Start.png"); //Assigning the data or file
  pEnd = loadImage("Pong_End.png"); //Assigning the data or file
```

I named my variable `timePlayed`.

22. Now we need to make the timer variable count up by one every time Processing runs through the draw loop. To do this, scroll down to the end of the draw loop and add the following code:

```
if (started == false) {  
    image(pStart, 0, 0, width, height); //start  
    println("Start");  
}  
  
timePlayed ++; //added to increment game timer  
}  
  
void serialEvent(Serial myPort) {
```

This line of code may seem a little confusing to some of you; this is because it is a shorthand form of code. Programmers often need to add the number one to a variable so they developed this shorthand for it. You can also write `timePlayed = timePlayed + 1;` but this way is faster. Also, make sure that you put this code just before the last curly bracket of the draw loop, otherwise your counter will only count up some of the time.

If you're really fancy, you can look up `millis()` and use that to make your counter add time, just make sure you still use a variable like `timePlayed` because eventually the players will want to reset the game, and there is no way to reset the values you get from `millis()`.

23. Now comes the tricky part. We want the variable `timePlayed` to affect the speed of the ball, and there are a couple different ways to do that. There is one way that involves creating a new variable (surprise!) and affecting that variable with `timePlayed` in order to change the vector of the ball. If you're bored, go ahead and do that instead of following along with the rest of this step. There's also an easier way to do it that involves simply adding a little to X and Y each time the ball bounces off the paddles. But we're going to take the middle path for now by simply adding a fraction of `timePlayed` each time Processing runs through the draw loop. In order to do this add this line below where you incremented your `timePlayed` variable:

```
pong.bRadius = pong.bRadius + (timePlayed * .001);
```

You can try multiplying `timePlayed` by a different number than `.001` if you want more or less of a challenge in your game play.

Here's what my code looks like now with that line added:

```
if (started == false) {
  image(pStart, 0, 0, width, height); //start
  println("Start");
}

timePlayed ++; //added to increment game timer
pong.bRadius = pong.bRadius + (timePlayed * .001); //added t
}

void serialEvent(Serial myPort) {
```

Lastly, we have to reset the `timePlayed` variable each time one of the players wins. However, you need to be careful exactly where this happens because what if the game is finished and there is a period of time between games? The ball would be bouncing really fast right at the beginning of the game, right?

This is what writing code is all about, creating something that sort of works and then making it better. Well, that and error messages. Don't freak out when you see error messages, they mean you're doing real coding and eventually you will come to rely on them to help you code.

Here's where I reset my counter and ball speed variables (because otherwise the second and third game would be very, very fast) in the Ball object's `reset` function, but there are a whole bunch of different places and ways you could do this:

```
void reset() {
  xpos = width/2;
  ypos = height/2;
  ydir = random(-1, 1);
  float dir = random(-1, 1);
  if (dir > 0) {
    xdir = 1;
  }
  else if (dir <= 0) {
    xdir = -1;
  }
  pong.xspeed = 8; //reset ball speed
  pong.yspeed = 2; //reset ball speed
  timePlayed = 0; //reset time played counter
  bRadius = 8.25; //reset ball vector
}
```

There are tons of different ways you could use the `timePlayed` variable; how else would you use it?

Blam. You just completed Pong using Arduino and Processing. Take a minute to play your version of Pong against someone. Show off your amazing reflexes and sweet coding skills. Stomp all Pong opposition. It is your game after all; you probably got really good at it while you tested your code!

At this point you should feel free to customize your game. Check out some basic graphical functions for Processing here:

<http://processing.org/learning/color/>

And

<http://processing.org/learning/pixels/>

Reflective Questions:

1. Nice! You just built a simple game and a simple game controller. What was your favorite part?
2. What was the most challenging part?
3. What other sensors could you use in place of the potentiometer?
4. How could you turn the breadboard into something that is more stable that you could use as a video game controller?

Challenges (these are tough!):

1. Can you replace one of the analog sensors with two buttons? This could probably be done easiest by creating a new function (or two), which checks to see if the buttons have been pressed, and then changes an integer variable up or down depending on which button has been pressed. Make sure this variable can only go as high as 1023 and as low as 0.
2. What are some other changes you could make to the game that depend on a button or keyboard press? How about changing the color of the players or background? What about adding a dial and a button so that you can control which aspect of the game you are changing the color of each time you press the button? Don't forget to add some text so the user knows which part of the game is being changed!
3. Can you change the ball's code so the faster the paddle is moving the faster the ball bounces off it? How about changing the action of the ball and paddle's bounce so that the angle is created using the speed of the paddle and the portion of the paddle hit?
4. For a really tough challenge: Can you make the game for four players, with two teams and two paddles on each side of the court? How about four players with four different teams, one on each side of the screen?