Lab 04: Bouncing Around the Screen

In this lab you will learn:

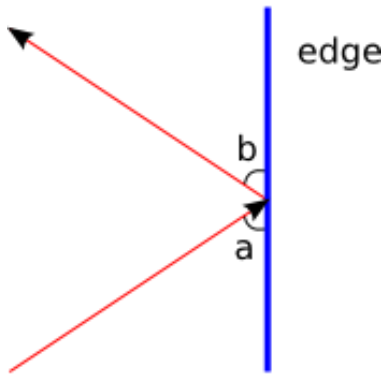
- How to make a ball move in any direction at a constant speed.
- Some basic uses of if-statements.
- How to perform simple actions when an edge is hit (such as stopping or expanding).

Introduction

Now let's create a program that makes a ball – or any shape or image – move in any direction and bounce off any edge of the screen.

To allow a ball to move in any direction we need to take into account its speed in both the x-direction and the y-direction. Thus we will be using two variables, `dx` and `dy`, to control the ball's speed. The variable `dx` is how many pixels the ball moves in the x-direction on each call to `draw()`, and `dy` is how many pixels it moves in the y-direction.

Another assumption we will make is that every time the ball hits an edge it bounces off at an out-going angle that is the same as the incoming angle, i.e. a = b in the following diagram:



Finally, we will ignore things like gravity and air friction. Such details can be added later.

Variables and Initialization

Let's walk through the creation of our bouncing-ball program step-by-step. First, we need a couple of variables to keep track of the position and the radius of the ball:

```
// Properties of the ball
float x;   // X position of the ball
float y;   // Y position of the ball
float r;   // Radius of the ball
```

And then we need two speed variables to keep track of how fast the ball moves along the x-axis and y-axis on each call to `draw()`:

```
float dx;  // Horizontal velocity of the ball
float dy;  // Vertical velocity of the ball
```

For convenience, let's also add a couple of colour variables:

```
// Convenience colour definitions
color white  = color(255);
color orange = color(255, 165, 0);
```

Note that all six of these variables are declared outside of any function, i.e. they are all global variables.

We initialize the screen and the four ball variables in `setup()`:

```
/**
 * Initialises the canvas and the ball variables.
 */
void setup() {
  size(500, 500);
  smooth();
  // initialise the ball
  x  = 250;  // start in the centre
  y  = 250;
  r  = 25;
  dx = 1;    // initial movement: down/right
  dy = 2;
}
```

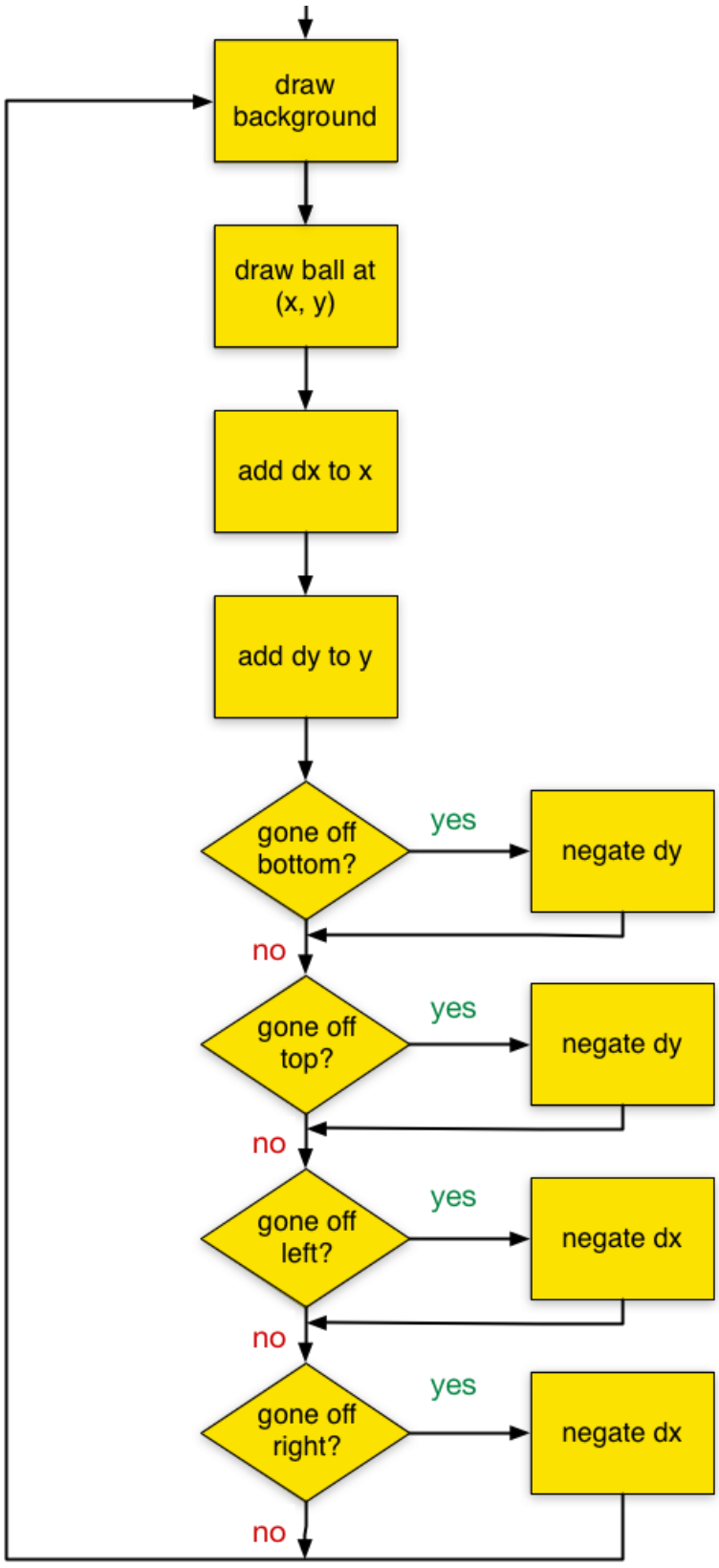
The precise starting values of `x`, `y`, `r`, `dx`, and `dy`, are, of course, up to you.

Animating the Ball

Now we need to think about how to animate the ball. The essential idea is to repeat the following steps:

- draw the background
- draw the ball at (x, y)
- move the ball to its next position by adding dx to x and dy to y
- check if the ball has hit one of the four edges, and, if so, change the direction of the ball by negating either dx or dy

The above way of describing a program is called pseudocode: it is a little bit like computer code, but written in English. We can also describe code using a flow chart. However, flow charts are not very common with modern programmers since they take up a lot more space than pseudocode without providing much (or any) more information.


draw background
draw ball at (x, y)
add dx to x
add dy to y
gone off bottom?
yes
negate dy
no
gone off top?
yes
negate dy
no
gone off left?
yes
negate dx
no
gone off right?
yes
negate dx
no

Following the flow chart, the actual Processing code we write uses four if-statements, one for each edge:

```
/**
 * Draws a single frame and animates the ball.
 */
void draw() {
  // clear the screen
  background(white);

  // draw the ball
  noStroke();
  fill(orange);
  ellipse(x, y, r*2, r*2);

  // move the ball
  x += dx;
  y += dy;

  // hit the left edge?
  if (x <= 0) {
    dx = -dx;
  }
  // hit the right edge?
  if (x >= width - 1) {
    dx = -dx;
  }
  // hit the top edge?
  if (y <= 0) {
    dy = -dy;
  }
  // hit the bottom edge?
  if (y >= height - 1) {
    dy = -dy;
  }
}
```

Notice the use of two variables `width` and `height`. These two variables are already declared by Processing and always reflect the width and height of the canvas. In our case, they will both have the value 500. If you change the parameters of the `size()` function in `setup()`, the value of these variables will change accordingly.

There are at least two advantages of using these variables. First, your code becomes more readable, as the if statement now clearly has something to do with the width and the height of the screen. Second, if you later decide to change your canvas size, you don't have to go back to this code and change the 499 into something different.

There is only a minor aspect to take into account. Since we have so far used 499, the right/bottom-most pixel coordinates in the if statements, but the variables `width` and `height` will each have the value 500 stored in them, we have to subtract 1.

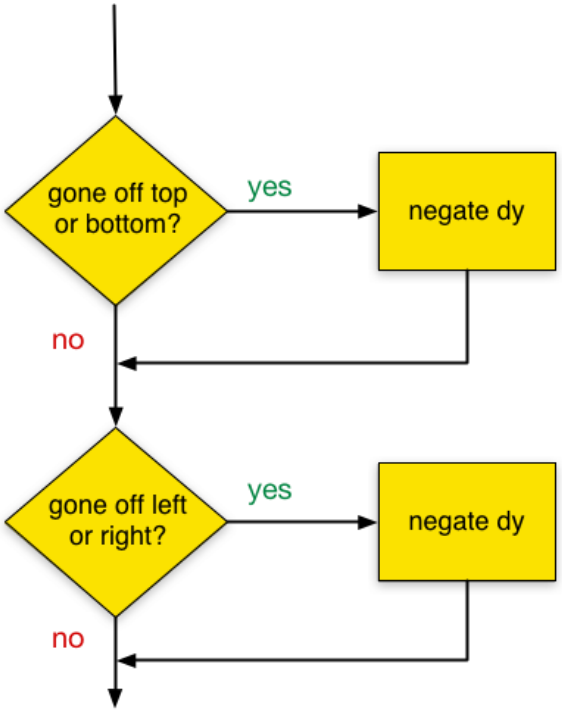
Another Way to Write the If-statements

There's often more than one correct way to write an if-statement. For instance, we can use || ("or") to combine the top/bottom test into one statement, and the left/right test into another:

```
// hit top or bottom edge?
if (y <= 0 || y >= height-1) {
   dy = -dy;
}

// hit left or right edge?
if (x <= 0 || x >= width-1) {
   dx = -dx;
}
```

The expression `y <= 0 || y >= height-1` is true when `y <= 0` is `true`, or when `y >= height-1` is `true`, or when both `y <= 0` and `y >= height-1` are `true` (which is impossible). The only time it is false is when both `y <= 0` is `false`, and `y >= height-1` is `false`. Here it is a flow chart:

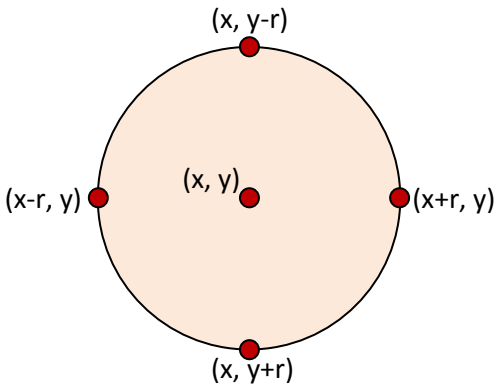


It might seem that two if-statements are preferable to four – it's less code after all. But it's not so simple. First, the two if-statement version has more complicated conditions. Second, the two if-statement version is less changeable. If you use this code in, say, a game, then it's likely that you'll want something different to happen to the ball depending on which edge it hits. Maybe the top edge is the sky and the bottom edge is the ground, and so the ball should fly off the screen if it hits the top, but bounce when it hits the bottom. You'll need one if-statement for each different kind of edge behaviour, as we wrote in the first program.

Bouncing off the Ball's Edge

The basic bouncing ball program can be changed in many different ways. For instance, right now the ball changes direction when its *centre point* hits an edge, which causes part of the ball to go off the screen.

Suppose, instead, we want the ball never to leave the screen when it bounces. Then we need to test if one of its four edge points has gone off the screen:



Recall that, by default, Processing sets the location on an ellipse by specifying its centre point, (x, y). So when the ball hits the right edge of the screen, we want it to reverse its x-direction as soon as its right-most point (x + r, y) hits the right edge. Similarly, we want the ball to reverse its y-direction when the point (x, y + r) hits the bottom edge.

Thus we can change the if-statements in draw to test for these four new points:

```
// hit the left edge?
if (x - r <= 0) {
  dx = -dx;
}
// hit the right edge?
if (x + r >= width - 1) {
  dx = -dx;
}
// hit the top edge?
if (y - r <= 0) {
  dy = -dy;
}
// hit the bottom edge?
if (y + r >= height - 1) {
  dy = -dy;
}
```

Make these changes in the program and run it. You'll see that it makes the ball and screen-edges seem solid.

Stopping at the Top

Let's set ourselves another problem: how can we make the ball stop moving completely when it hits the top of the screen?

It turns out to be a simple change:

```
// hit the top edge?
if (y - r <= 0) {
  dx = 0;
  dy = 0;
}
```

We modify the code in the body of the if-statement that checks for hitting the top edge to set the speed of the ball to be 0 in both the x-direction and y-direction. This cases the ball to stop dead as soon as it hits the top edge of the screen.

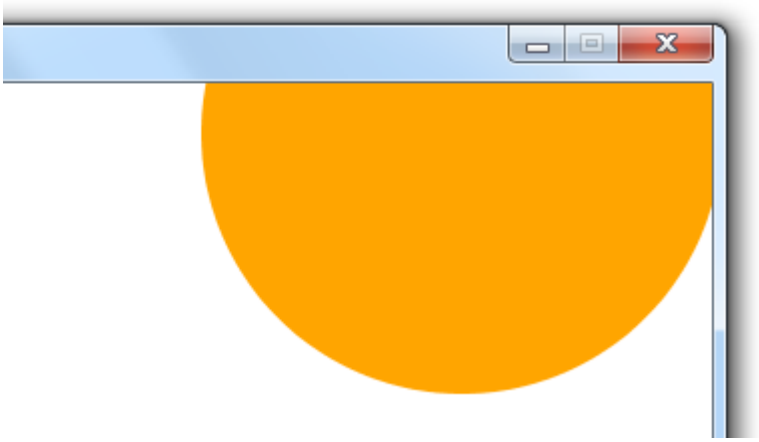
Keep in mind that even though the ball is not moving, `draw()` is still being called. The ellipse is still being drawn, and x and y are still being updated. But its position never changes because `dx` and `dy` are both 0.

Expanding when the Ball Hits the Top

How can we make the ball bigger when it hits the top of the screen? In other words, we want the diameter of the ball to get bigger every time hits the top edge. Again, this is a relatively simple modification:

```
// hit the top edge?
if (y - r <= 0) {
  dy  = -dy;
  r  += 5;
}
```

Make this change and run the program, and you should see that the ball gets a little bit bigger each time it hits the top edge.



Debugging: Finding a Bug

Unfortunately, the above change doesn't quite work. When you run the modified program, the ball most likely gets stuck on the top edge of the screen and inflates to a huge size without ever bouncing off any other edges. Obviously something is wrong with the program. But what?

This is a tricky error to deal with because there is no obvious flaw in any line of our code. The fact that the problem only occurs when the ball hits the top edge of the screen gives a hint about where the problem might be:

```
// hit the top edge?
if (y - r <= 0) {
  dy  = -dy;
  r  += 5;
}
```

To figure out what is going on, it is helpful to know what lines of code are being called. A simple way to check this is to use a `println` statement to print a message every time the code in the if-statement body is called:

```
// hit the top edge?
if (y - r <= 0) {
  println("hit top edge?");
  dy  = -dy;
  r  += 5;
}
```

`println` prints a message in the black window at the bottom of the Processing editor known as the **console window**. We usually use `println` to help us understand and fix programs.

When you run the code after inserting this `println`, the ball still gets stuck and inflates at the top of the screen. But you should now see this in the black window at the bottom:

```
hit top edge?
hit top edge?
hit top edge?
hit top edge?
```

What this tells us is that the body of the top-edge if-statement is being called again and again. Somehow, the top point of the ball has gotten stuck above the top edge of the screen and can't get out.

How might this happen? Let's try one more little trick: lets print the values of `y`, `dy`, and `r` the first time the top-edge if-statement body is called, and then lets immediately stop the program using `System.exit(0)` to see the results:

```
if (y - r <= 0) {
  println("hit top edge?");
  println("y = " + y);
  println("r = " + r);
  println("dy = " + dy);
  System.exit(0);  // immediately stops the program

  dy  = -dy;
  r  += 5;
}
```

Here's the output that is produced on the console when the program runs:

```
hit top edge?
y = 24.0
r = 25.0
dy = -2;
```

This tells us the value of all the relevant variables the first time that `y - r <= 0` evaluates to `true`. The expression `y - r` is less than 0 because y is 24 and `r` is 25.

So that makes sense. Now let's imagine what happens when the next two statements are run:

```
dy  = -dy;
r  += 5;
```

After running these, `dy` is 2, `r` is 30, and `y` is still 24. This is where the problem lies: the expression `y - r <= 0` is `true` because if you substitute the values for their variables, you get the true expression `24 - 30 <= 0`. So the point we use to test if the ball has gone off the top edge of the screen is already off the top edge!

The next time `draw()` is called, y gets incremented by 2 (by `y += dy`), and so is now 26. Thus `y - r <= 0` is true (because it is 26 - 30 <= 0), and the top-edge if-statement body is executed once more. When that runs (ignoring `System.exit(0)` for now), `r` becomes 35 and `dy` becomes -2. The next time `draw()` is called, `y` is back to 24 and `r` is 35, and so `y - r <= 0` is still `true`. And so on…

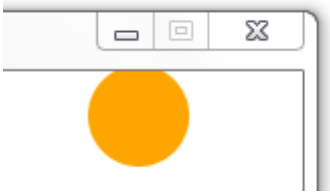
So `y` flips back and forth between 24 and 26 on each call to `draw()`. Since `r` also increases on each call, the expression `y - r <= 0` remains true. Thus the ball is trapped off the top edge of the screen.

This is a tricky problem to catch. To understand it, you need to trace through a few calls to `draw()` to see that the values of y are stuck at 24 and 26. These sorts of errors get easier to catch with experience, but you can never be rid of them completely: dealing with subtle bugs is one of the realities of programming.

Debugging: Fixing the Problem

So we've identified the problem, but how do we fix it? This is not completely obvious because there are many ways that we could modify the value of `y` (or even `dy`) that might fix it.

But let's try to understand a little more what is going on. The first time that the top-edge if-statement condition is true, the image on the screen looks something like this:



The top of the ball goes a little bit into the top edge of the screen, i.e. the ball interpenetrates the edge. The ball and the edge are supposed to be hard surfaces that don't allow any sort of interpenetration. But our arithmetic allows for them to intersect, and so we have to do something to stop that.

The basic idea is that when the ball interpenetrates and edge, we will move it back a little bit so it is just touching the edge. For example:

```
// hit the left edge?
if (x - r <= 0) {
  dx = -dx;
  x  = r;
}
```

In this if-statement, when the ball hits the left edge of the screen we reverse its direction, and then set x so that the ball is just touching the edge without going over. In a sense, we are re-setting the position of the ball every time it hits an edge.

We have to reset the ball's position for each if-statement:

```
// hit the left edge?
if (x - r <= 0) {
  dx = -dx;
  x  = r;
}
// hit the right edge?
if (x + r >= width - 1) {
  dx = -dx;
  x  = width - 1 - r;
}
// hit the top edge?
if (y - r <= 0) {
  dy = -dy;
  r += 5; // make ball larger
  y  = r;
}
// hit the bottom edge?
if (y + r >= height - 1) {
  dy = -dy;
  y  = height - 1 - r;
}
```

This works! Run it and you will see that the ball now stays on the screen and expands when it hits the top.

Introducing the `random()` Function

The code for this example increments the size of the ball a fixed amount each time, but what if we wanted to increase it a random amount? Processing has a built in function for calculating a random number that is easy to use. Change the code for hitting the top edge to:

```
if (y - r <= 0) {
  dy  = -dy;
  r  += random(10);
  y   = r;
}
```

Also introduce the following change:

```
if (y + r >= height - 1) {
  dy = -dy;
  y  = height - 1 - r;
  orange = color(random(255),random(255),random(255));
}
```

Question: Before running the program, what do you think that this second code change will do? What do you then think about the name "`orange`" as a variable name?

Theory Quiz

Before attempting the below programming tasks make sure that you have a go at the theory test on AUTonline. A theory test is given each week and is just a means for you to gauge how much you have remembered from the tutorial material. The number of quizzes you complete will be tracked to determine how much effort you are putting in to learning programming.

Programming Questions

1. Modify the bouncing ball program so that a small ellipse is drawn under the ball as a shadow to give the illusion of 3-dimensions. For example:
   Make sure the shadow gets bigger as the ball's size increases.
2. Modify the bouncing ball program so that when the ball hits the left edge of the screen, its speed increases by a factor of 1.5. Also, keep the ball's size fixed: do not have it change size when it hits an edge.
3. Write a program that makes a square bounce around the screen. When the squares hit an edge it should hit the edge exactly, without any part of it going off the screen. Keep the dimensions of the square the same throughout the program (e.g. don't let it get bigger when it hits an edge).
4. Write a program that makes an image (i.e. a `PImage`) bounce around the screen. When the image hits an edge it should hit the edge exactly, without any part of it going off the screen.
5. Modify the bouncing ball program so that the ball changes to red when it hits the top edge; to green when it hits the right edge; to blue when it hits the bottom edge; and to orange when it hits the left edge. Also, keep the ball's size fixed: do not have it change size when it hits an edge.
6. Draw a 200-by-200 square centred in the middle of a 500-by-500 screen. Write a program that makes a ball bounce inside this square.
7. Make two different coloured balls bounce around the screen. You'll need to keep track of the position, velocity, and colour of each ball, so your program will need at least 10 different variables.
8. Modify the bouncing ball program so that the ball's diameter does not change when it hits an edge. Instead, use the map function to make the balls diameter go from 25 to 150 as the mouse pointer moves horizontally across the screen.
   For example, when the mouse pointer is halfway across the screen, the ball should have a diameter of about 87 (i.e. halfway between 25 and 150). When the mouse pointer is at the very left of the screen, the ball's diameter should be 25.
9. Modify the bouncing ball program so that every fifth time the ball hits any edge it changes colour. For example, it can start out orange, and the after 5 edge hits it turns red. Then after 5 more edge hits it turns red, and so on. Also, keep the ball's size fixed: do not have it change size when it hits an edge.