Bouncing Chaos

Mathematically, chaos can be thought of as 'sensitivity to initial conditions'. In other words, even in a completely deterministic scenario (that is, one governed entirely by predictable rules, not randomness or chance), even tiny modifications in the starting conditions can have enormous repercussions for the predictability of the outcomes.

Real life chaotic systems include the weather (think of the classic 'butterfly effect') and the stock market. There are many classic examples of apparently simple chaotic systems, including the double pendulum (a pendulum suspended from a pendulum) where the eventual motion appears random simply because it is so complex.

Ball in a bowl

We're going to simulate dropping a ball into a hemispherical bowl. Whenever the ball lands on the curved edge of the bowl, it will bounce. By creating a bunch of balls, each with a very slightly different starting configuration (varying either initial position or initial velocity) we can see how different the results would have been for our bouncing ball had it been released under very slightly different conditions. The inspiration for this project came from the mathematical animations of Matt Henderson.

The ball class

When we first learned how to use vectors in p5, we created a *Ball* class. We'll do something similar here:

```
class Ball{
  constructor(x, y, vx, vy){
    this.pos = createVector(x, y);
    this.vel = createVector(vx, vy);
    this.acc = createVector(0, 0.04);
  }
  update(){
    this.vel.add(this.acc);
    this.pos.add(this.vel);
  }
  display(){
    circle(this.pos.x, this.pos.y, 20);
  }
}
```

This code forms the basis of many moving objects. Essentially we keep track of position, velocity and acceleration, updating the position using the current velocity and updating the velocity using the current acceleration. In this case, acceleration is fixed in the down-screen direction to simulate gravity.

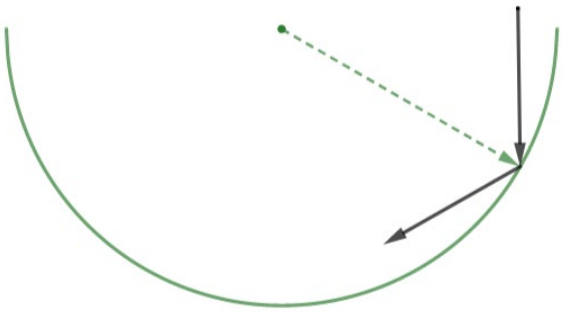
```
let balls = [];

function setup() {
  createCanvas(400, 400);
  balls.push(new Ball(width/2, height/2, 0, 0));
}

function draw() {
  background(220);
  for (let ball of balls){
    ball.update();
    ball.display();
  }
}
```

After setting up the bare bones of our *Ball* class, we create a ball to make sure it behaves as we want. I define the *balls* array in the main program scope, then *push* a new *Ball* object into the array within *setup*.

Finally, every time the *draw* loop runs, we loop through all balls in the array, update them and display them.

Reflecting vectors

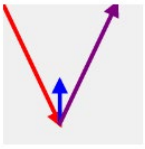
Last time we played with vectors, we added a bit of code which checked to see if the ball hit a boundary, and made it bounce. This time, the bounce will be a little more complicated because the angle at which the ball hits the curved boundary will determine the angle at which it rebounds:



When a ray of light, or an object for that matter, hits a surface, it bounces off in such a way that the component of velocity it had which was normal to the surface is reversed, but the component tangential to the surface is unchanged. That's because, ignoring friction, the surface doesn't impede motion parallel to itself, but forces a reversal perpendicular to itself.

Because the normal to a semicircle always passes through the centre, we can essentially reflect in that vector. P5 has a *reflect* function built in, so we don't have to faff around with matrices (if you want an extra challenge, give it a go: you'll need to remember to translate, then you can use the standard matrix for reflecting in a line through the origin).

To use p5's *reflect* vector method we need to understand how it works. The online reference is a good place to start: https://p5js.org/reference/#/p5.Vector/reflect



edit reset copy

```
function draw() {
  background(240);

  let v0 = createVector(0, 0);
  let v1 = createVector(mouseX, mouseY);
  drawArrow(v0, v1, 'red');

  let n = createVector(0, -30);
  drawArrow(v1, n, 'blue');

  let r = v1.copy();
  r.reflect(n);
  drawArrow(v1, r, 'purple');
}
```

In this animation, the red arrow is reflected in the blue arrow to produce the purple arrow.

Note that *drawArrow* is not a built-in function – it presumably would have been defined elsewhere in the code – but it is fairly self-explanatory.

Provided we construct our normal vector *pointing towards the centre of the circle* rather than away from it, we should be able to simply reflect our velocity vector in it whenever the ball reaches the curved surface.

To do list:

- Build a way for a ball to see if it has reached the boundary (a simple distance function will do, given that our boundary is circular in shape). If it reaches the boundary:
  - Construct the normal vector from the ball's position to the centre of the circle.
  - Reflect the ball's velocity vector in that normal vector.

The distance checker

```
bounce(){
  let [x, y, R] = [width/2, height/2, width*0.45];
  return (this.pos.x - x)**2+(this.pos.y - y)**2 > R**2;
}
```

This is a method within the *Ball* class, and will simply return *True* if the ball is further from the centre of the screen than *R* (which is just under half the screen width).

The reflection

```
update(){
  if (this.bounce()){
    let n = createVector(width/2 - this.pos.x, height/2 - this.pos.y);
    this.vel.reflect(n);
  }
  this.vel.add(this.acc);
  this.pos.add(this.vel);
}
```

Within the *Ball* class still, I've added a clause to the beginning of the *update* method which reflects the velocity in the case where the ball needs to bounce.

Note that the normal vector *n* is a vector from my own position to the screen's centre.

Adding the bowl

If you try running the script now (maybe changing the initial velocity so it's more exciting than simply bouncing up and down vertically), you should find that the ball displays the right kind of behaviour. To make it more obvious what's going on, however, we might want to draw in the semicircle itself. We could just draw a full circle (if we give our ball enough oomph at the start, it'll bounce off the circle at any point, after all), but it's up to you.

```
function draw() {
  background(220);
  noFill();
  stroke(120);
  arc(width/2, height/2, width-10, height-10, 0, PI);
  for (let ball of balls){
    ball.update();
    ball.display();
  }
```

I just want the half-circle line drawn in, so I've included a *noFill()* call before drawing an *arc* object. Think of this like drawing an ellipse: give the coordinates of the centre, then the width and height, and finally the parameters for the start and stop angles (measured clockwise from the positive x direction).

Note that I've set this to be 10 pixels shy of the full width. This will give the illusion that the ball is bouncing exactly on the surface even though in reality the centre of the ball is bouncing on a slightly smaller circle.

Also note that, now I've called *noFill()* during the *draw* loop, the ball will be displayed without fill. You can change that in the main *Ball* code's *display* method if you like:

```
  display(){
    noStroke();
    fill("teal");
    circle(this.pos.x, this.pos.y, 20);
  }
}
```

As a general rule, it's a good idea to specify both *fill* and *stroke* whenever you display an object, since otherwise the settings will carry over from the last time they were changed in your code.

More balls!

All that remains to simulate chaos is to add more balls to our array, each with slightly different starting parameters:

```
let balls = [];
let numBalls = 20;

function setup() {
  createCanvas(400, 400);
  for (let i=0; i<numBalls; i++){
    balls.push(new Ball(width/2, height/2, 2+0.01*i, 0));
  }
}
```

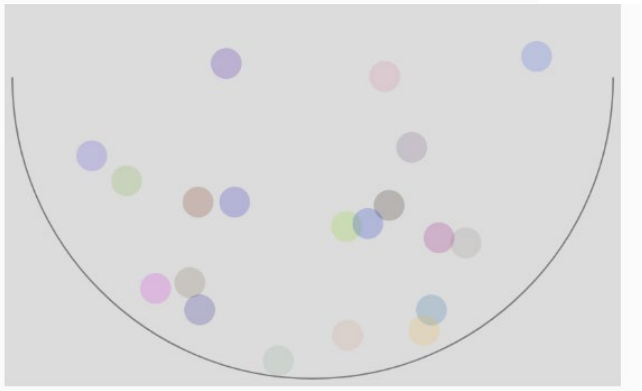
I have set a variable *numBalls* so we can more easily modify this setting.
The loop in *setup* gives each ball a slightly different velocity, from 2 to 2.2 pixels per frame. You can make this even smaller if you like. Eventually the path of the balls will diverge.

Even better if…

This does demonstrate what we want, but there are a few improvements you might like to make. For a start, it's not very easy to see the different balls because they're all the same colour. The easiest solution to this is to set each to a random colour. Maybe modify the opacity through the *alpha* value so they are slightly see-through:

```
class Ball{
  constructor(x, y, vx, vy){
    this.pos = createVector(x, y);
    this.vel = createVector(vx, vy);
    this.acc = createVector(0, 0.04);
    this.color = color(random(255),random(255),random(255), 50);
  }
```

The lower the fourth value (*alpha*) is set, the more transparent the colour. Remember to also update the *display* method to *fill(this.color)*.



Showing a trail

By making use of another feature we learned about recently, we can have each ball leave a trail so we can see where it's been. So as not to tax the computer too heavily, we should build in a maximum trail length, too. We'll need to update the *constructor* method…

```
class Ball{
  constructor(x, y, vx, vy){
    this.pos = createVector(x, y);
    this.vel = createVector(vx, vy);
    this.acc = createVector(0, 0.04);
    this.color = color(random(255),random(255),random(255), 50);
    this.trail = [this.pos.copy()];
  }
```

This initialises the trail as an attribute of the class.

… and the *update* method…

```
update(){
  if (this.bounce()){
    let n = createVector(width/2 - this.pos.x, height/2 - this.pos.y);
    this.vel.reflect(n);
  }
  if (this.trail.length > 100){this.trail.shift()}
  this.trail.push(this.pos.copy());
  this.vel.add(this.acc);
  this.pos.add(this.vel);
}
```

If the trail is too long, *shift* is used to remove the oldest entry, and then the latest position is pushed into the array. Note the use of *copy* so that the array isn't just full of pointers to the current position.

…and the *display* method…

```
display(){
  stroke(this.color);
  noFill();
  beginShape();
  for (let i=0; i<this.trail.length; i++){
    vertex(this.trail[i].x, this.trail[i].y);
  }
  endShape();
  noStroke();
  fill(this.color);
  circle(this.pos.x, this.pos.y, 20);
}
```

Remember to set *noFill* and *stroke* so that we just get lines from the first to last vertex.