# Adam Brookes' Elastic Collision Code

### Introduction to the code

Canvas collision consists of 5 classes, so I'll just a brief outline of what each class does.

## Ball.js

This class holds the ball object. A ball contains:

- position - *a vector used to store the x and y coordinates of the **centre** of the ball*
- lastGoodPosition – *a vector used to store the position of the ball before a collision*
- velocity – *a vector used to represent the x and y values of the balls velocity*
- radius – *a **var** that represents the radius of a ball*
- mass - *a **var** that represents the mass of the ball*
- colour – *a hex color value*

The only functions the ball class contains are the getters and setters for its member variables.

## canvas.js

As the name says this class loads the HTML5 canvas. It can also be thought of as a manager that calls the appropriate functions from the Renderer and Simulation classes. It is in charge of creating the ball objects, and is where the mainLoop runs the program.

## Simulation.js

This class manages all simulation aspects of the program. It has the array of balls passed into the Simulation update method. From here it checks if the ball has collided with a wall, and checks for balls colliding with each other.

## Renderer.js

This class is in charge of rendering all of the balls and the canvas background using the canvas API.

## Vector.js

This is a 2D vector class with vector functions including:

- Add
- Subtract
- Multiply
- Divide
- Normalise
- Magnitude
- Dot Product

If you do not know the basics of using vectors try reading about it at wolfram ( http://mathworld.wolfram.com/Vector.html ). For solving collision problems you will need to understand:

how to add and subtract vectors,
find unit vectors,
multiply a vector by a scalar, and
compute a dot product.

If you are not familiar with vectors, this might sound hard, but keep in mind that all you really need is addition, subtraction, multiplication, division and taking square roots; no complex trigonometry.

## Moving the balls

Moving the balls happens in the Simulation class and the code looks as follows:

```
1.  function updateBallPos(deltaTime, ballArray) {
2.      for (var i = 0; i < ballArray.length; i++) {
3.              ballArray[i].lastGoodPosition = ballArray[i].position;
                // save the balls last good position.
4.              ballArray[i].position =
    ballArray[i].position.add((ballArray[i].velocity.multiply(deltaTime/10)));
5.              // above add the balls (velocity * deltaTime) to position.
6.      } // end for
7.  } // end function
```

In this code we first store the position of the ball into a lastPosition variable. This is done so that when a collision is detected we can move the ball back to the place it was before the collision and work out of the collision response from there. This avoids objects being trapped in each other.

We then move the ball; this is case of adding the velocity vector to the position vector. We multiply the velocity by the deltaTime to give the balls processor independent animation.

## Detecting a ball collision

Before we can work out the collision response between two balls, we need to know when two balls have collided. This done using Pythagoras theorem, $a^2 + b^2 = c^2$ to work out the distance between the two circles centers.

We do not immediately know the lengths of sides a and b, but we do know position of each ball so we can easily work that out by subtracting ball 1′s X position from ball 2′s Y position and ball 1 X's position with ball 2′s Y position.

We can then solve C with a bit of algebraic rearrangement. So C = square root($a^2 + b^2$). This gives us the distance between the centers of the two balls.
We then need to check this value against the sum of the two radii, if distanceBetween is less than the sum of the two radii the balls have collided.

```
1.  function checkBallCollision(ball1, ball2) {
2.      var xDistance = (ball2.getX() - ball1.getX());
3.      var yDistance = (ball2.getY() - ball1.getY());
4.      var distanceBetween = Math.sqrt((xDistance * xDistance) + (yDistance
    *yDistance));
5.      var sumOfRadius = ((ball1.getRadius()) + (ball2.getRadius()));
                // add the balls radius together
6.  if (distanceBetween < sumOfRadius) {
7.          return true;
8.   } else {
9.          return false;
10. } // if/else
11. } // function end
```

## Understanding the theory of elastic collision

An elastic collision is a collision which kinetic energy is conserved. That means there is no energy lost as heat or sound during the collision. In the real word there are no perfectly elastic collisions on an everyday scale of size, but you can get a sense of elastic collision by imagining a perfect pool ball, which doesn't waste any energy when it collides.

## Five steps of working out elastic collision response

The best way to demonstrate the responses we are trying to calculate is using the diagram depicted at this Wikipedia page.

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

We can calculate the collision response in five steps.

### Step 1.

Firstly we need to find the **unit normal** and **unit tangent** Vectors.

The **unit normal** vector has a magnitude of 1 and a direction that is perpendicular to the surfaces of the objects at the point of collision. To get this we find the difference between the coordinates at the center of both circles. And then normalize that vector to find the unit vector ( this is done by dividing by the magnitude of the normal vector).

```
1         var xDistance = (ball2.getX() - ball1.getX());
2         var yDistance = (ball2.getY() - ball1.getY());
3
4         var normalVector = new vector(xDistance, yDistance);
5         normalVector = normalVector.normalise();
```

The **unit tangent** vector also has a magnitude of 1 but is tangent to the circles' surfaces at the point of collision. This is easy to find out from the unit normal vector. You just the make the x component of the **unit tangent** vector equal to the negative y component of the unit normal vector, and make the y component of the **unit tangent** vector equal to the x component of the unit normal vector.

```
var tangentVector = new vector((normalVector.getY() * -1), normalVector.getX());
```

### Step 2.

Next we need to resolve the velocity vectors into normal and tangential components. To do this we project the velocity vectors by taking the dot product of the velocity vectors with the unit normal and unit tangent vectors.

```
1  // create ball scalar normal direction.
2         var ball1scalarNormal =  normalVector.dot(ball1.velocity);
3         var ball2scalarNormal = normalVector.dot(ball2.velocity);
4  // create scalar velocity in the tagential direction.
```

```
5        var ball1scalarTangential = tangentVector.dot(ball1.velocity);

6        var ball2scalarTangential = tangentVector.dot(ball2.velocity);
```

### Step 3.

We now need to find the new normal velocities, and to do this we look at the collision in one dimension. The velocities of the two balls along the normal direction are perpendicular to the surfaces of the circles at the point of collision so this really is a one-dimensional collision.
The next code is based on the theory that the **kinetic energy** of an object is one-half times its mass times the square of it's velocity. And the **momentum** of an object is the dot product of it's mass and velocity.

```
1. var ball1ScalarNormalAfter = (ball1scalarNormal * (ball1.getMass() -
   ball2.getMass()) +
2. 2 * ball2.getMass() * ball2scalarNormal) / (ball1.getMass() + ball2.getMass());
3. var ball2ScalarNormalAfter = (ball2scalarNormal * (ball2.getMass() -
   ball1.getMass()) +
4. 2 * ball1.getMass() * ball1scalarNormal) / (ball1.getMass() + ball2.getMass());
```

### Step 4.

The next step is to convert the scalar normal and tangential velocities into vectors. To do this we just multiply the unit normal vector by the scalar normal velocity, and you end up with a vector which has a direction normal to the surface at the point of collision and which has a magnitude equal to the normal component of the velocity. The same happens for the tangential component.

```
1. var ball1scalarNormalAfter_vector = normalVector.multiply(ball1ScalarNormalAfter);
2. var ball2scalarNormalAfter_vector = normalVector.multiply(ball2ScalarNormalAfter);
3. var ball1ScalarNormalVector = (tangentVector.multiply(ball1scalarTangential));
4. var ball2ScalarNormalVector = (tangentVector.multiply(ball2scalarTangential));;
```

### Step 5.

Lastly you can find the final velocity vectors by adding the normal and tangential components for each object.

```
1. ball1.velocity = ball1ScalarNormalVector.add(ball1scalarNormalAfter_vector);
2. ball2.velocity =ball2ScalarNormalVector.add(ball2scalarNormalAfter_vector);
```