# Creating the Motion Trail

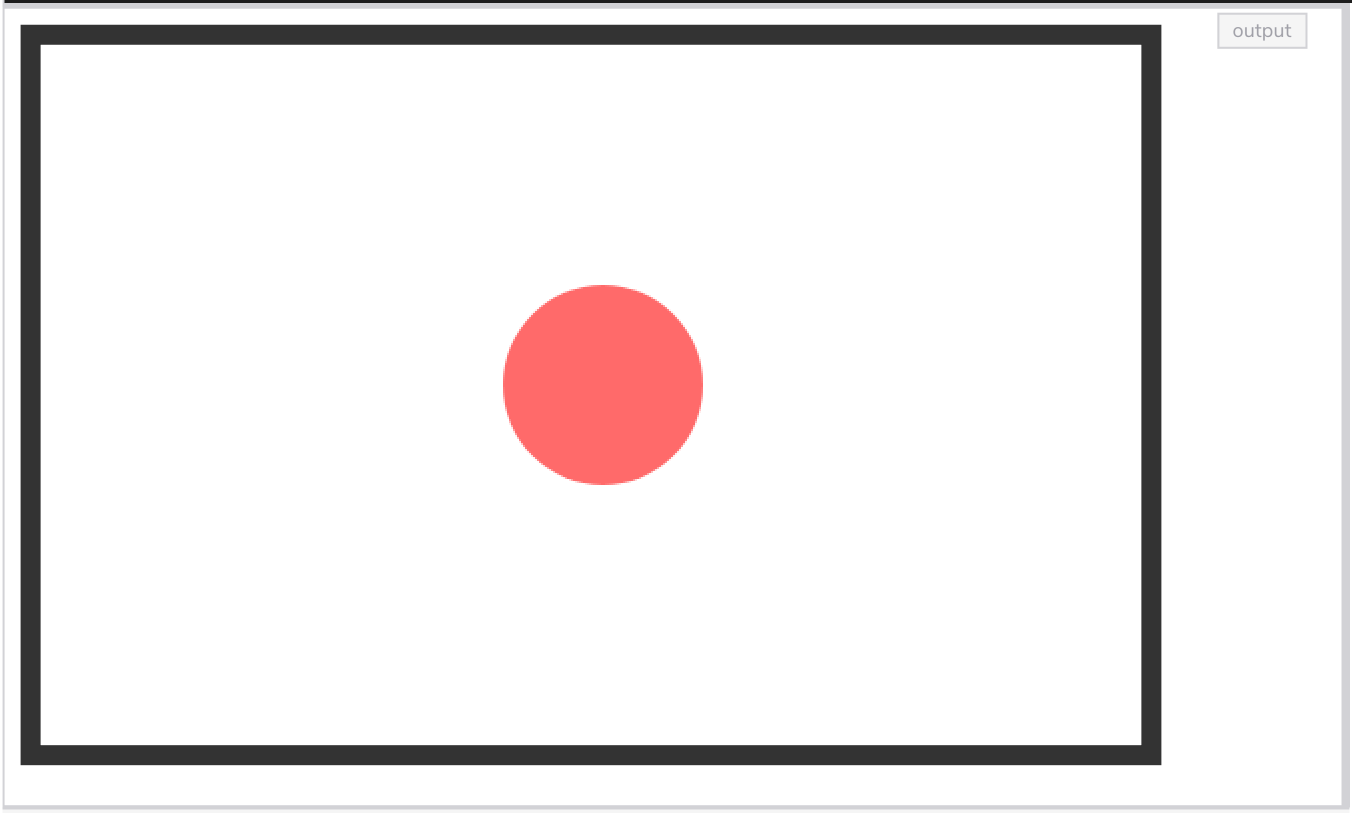
WE'LL COVER THE FOLLOWING

The task ahead of us is pretty straightforward - especially given what we looked at in the previous section. What we are going to do now is take an object that moves around the `canvas` and give it a motion trail. You can create your own starting point for this, but if you want to closely follow along, continue with our usual example where we have a `canvas` element with an `id` of **myCanvas**. Inside that HTML document, add the following into the `script` tag:

HTML JavaScript

javascript

```javascript
var canvas = document.querySelector("#myCanvas");
var context = canvas.getContext("2d");

var xPos = -100;
var yPos = 170;

function update() {
  context.clearRect(0, 0, canvas.width, canvas.height);

  context.beginPath();
  context.arc(xPos, yPos, 50, 0, 2 * Math.PI, true);
  context.fillStyle = "#FF6A6A";
  context.fill();

  // update position
  if (xPos > 600) {
    xPos = -100;
  }
  }
  xPos += 3;

  requestAnimationFrame(update);
}
update();
```



Once you've added this code, go ahead and preview what you have in your browser. If everything worked out fine, you should see a circle moving from left to right. Right now, this circle shows no motion trail. We are going to fix that right up in the next couple of sections.

## Understanding How Things Get Drawn #

The first thing to do is to get an idea of how things are getting drawn to the screen. In our example, we have the `update` function that is part of the `requestAnimationFrame` loop. Inside it, the following code is responsible for drawing our circle:

```
context.beginPath();
context.arc(xPos, yPos, 50, 0, 2 * Math.PI, true);
context.fillStyle = "#FF6A6A";
context.fill();
```

The `xPos` and `yPos` variables are responsible for figuring out where our circle is positioned. Just a few lines below our drawing code, we have the following:

```
// update position
if (xPos > 600) {
  xPos = -100;
}
xPos += 3;
```

This code is responsible for two things. The first is resetting the value of `xPos` if it gets larger than 600. The second is incrementing the value of `xPos` by 3 each time `requestAnimationFrame` calls our `update` function. In other words, around 60 times a second ideally.

You put all of this together, you can see why our circle moves the way it does. It starts off at -100, and makes its way right by 3 pixels each time our frame is updated. Once our `xPos` value gets larger than 600, the `xPos` value gets reset to -100 which causes our circle's position to be reset as well.

## Storing our Source Object's Position #

Now we get to the good stuff. This is the part where we specify how big our motion trail is going to be and create our array that stores the position of our source object. Above your `update` function, add the following code:

```
var motionTrailLength = 10;
var positions = [];

function storeLastPosition(xPos, yPos) {
  // push an item
  positions.push({
    x: xPos,
    y: yPos
  });

  //get rid of first item
  if (positions.length > motionTrailLength) {
    positions.shift();
  }
}
```

This `motionTrailLength` variable specifies how long our motion trail is going to be. The `positions` array stores the x and y values of our source object. The storeLastPosition function is responsible for ensuring our positions array is

no longer than our motion trail's length. This is where the queue logic we looked at earlier comes into play.

Just adding this code isn't enough. We need to actually store our source object's position. For that, we go back to our `update` function and make a call to `storeLastPosition` just after we draw our circle. Go ahead and add line 9:

```
function update() {
  context.clearRect(0, 0, canvas.width, canvas.height);

  context.beginPath();
  context.arc(xPos, yPos, 50, 0, 2 * Math.PI, true);
  context.fillStyle = "#FF6A6A";
  context.fill();

  storeLastPosition(xPos, yPos);

  // update position
  if (xPos > 600) {
    xPos = -100;
  }
  xPos += 3;

  requestAnimationFrame(update);
}
```

This ensures that immediately after we draw our circle at its new position, we store that position in our `positions` array. There is a subtle detail I want you to pay attention to. Notice the order we are doing things in. We first draw our circle using the latest values from `xPos` and `yPos`. After the circle gets drawn, we store that position using the `storeLastPosition` function. Keep this in mind, for we will revisit this in a few moments.

## Drawing the Motion Trail #

We are now at the last and (possibly) most tricky step. It is time to draw our motion trail. What we are going to do is go through our `positions` array and draw a circle using the co-ordinates stored at each array entry.

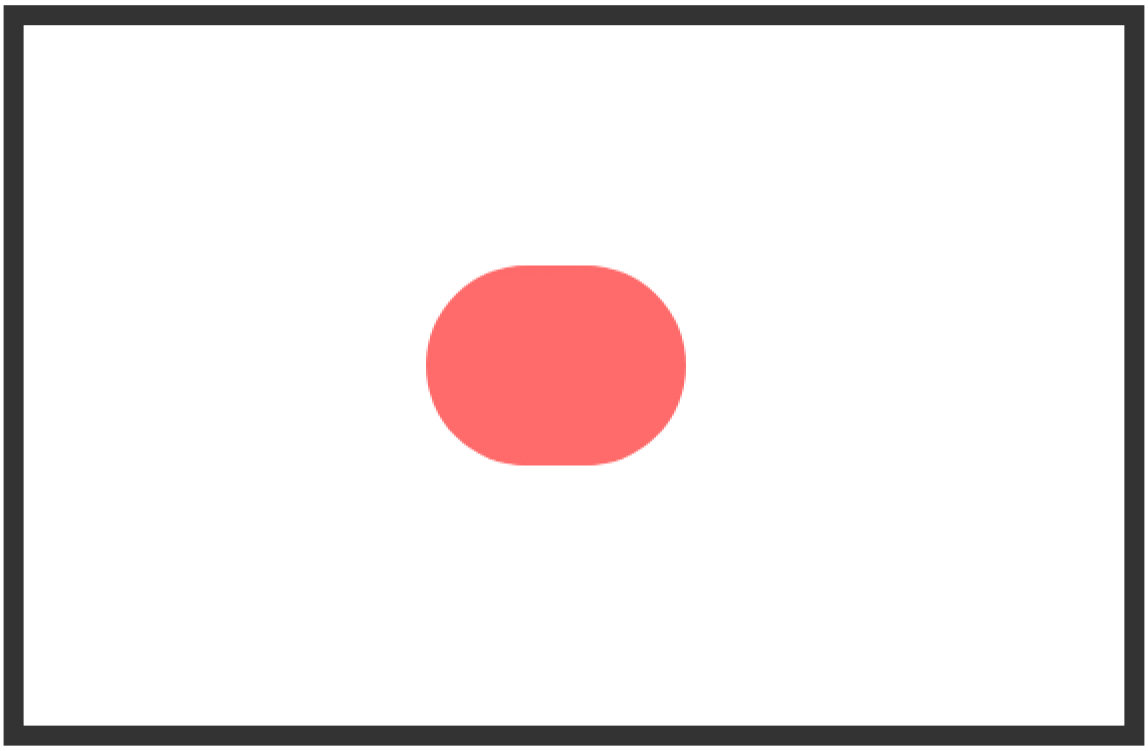
Inside your `update` function, just below the `clearRect` call, add lines 13-18 to your code:

HTML JavaScript

javascript

```javascript
var canvas = document.querySelector("#myCanvas");
var context = canvas.getContext("2d");

var xPos = -100;
var yPos = 170;

function update() {
  context.clearRect(0, 0, canvas.width, canvas.height);

  for (var i = 0; i < positions.length; i++) {
    context.beginPath();
    context.arc(positions[i].x, positions[i].y, 50, 0, 2 * Math.PI, true);
    context.fillStyle = "#FF6A6A";
    context.fill();
  }

  context.beginPath();
  context.arc(xPos, yPos, 50, 0, 2 * Math.PI, true);
  context.fillStyle = "#FF6A6A";
  context.fill();

  storeLastPosition(xPos, yPos);

  // update position
  if (xPos > 600) {
    xPos = -100;
  }
  xPos += 3;

  requestAnimationFrame(update);
}
```

output



You’ll see our circle sliding from left to right. You’ll also see the motion trail.

Our motion trail is literally a direct copy of our source object. The only difference is that each of our source object look-a-likes are positioned a few pixels in the past. You can see why by looking at the code you just added:

```
for (var i = 0; i < positions.length; i++) {
  context.beginPath();
  context.arc(positions[i].x, positions[i].y, 50, 0, 2 * Math.PI, true);
  context.fillStyle = "#FF6A6A";
  context.fill();
}
```

You can see that we simply copied the earlier drawing code for our source object, placed it all inside a `for` loop that goes through the `positions` array, and specified that the x/y position for our circle comes from values stored inside our `positions` array. As you just saw when you previewed in your browser, our code creates a motion trail in letter, but it doesn't quite create it in spirit! We don't want that.

Let's first adjust our motion trail by having the circles fade away the further away in the motion trail you go. We can do that easily by using some simple array length shenanigans and a RGBA color value. Modify our `for` loop by making changes to line 2 and 6:

```
for (var i = 0; i < positions.length; i++) {
  var ratio = (i + 1) / positions.length;

  context.beginPath();
  context.arc(positions[i].x, positions[i].y, 50, 0, 2 * Math.PI, true);
  context.fillStyle = "rgba(204, 102, 153, " + ratio / 2 + ")";
  context.fill();
}
```

The changes we made allow your circles to fade away the further from the source object they are. The `ratio` variable stores a number between `1 / positions.length` (when `i` is equal to 0) and 1. This range is based on the result of dividing `i + 1` with the length of our `positions` array.

This `ratio` value is then used in the `fillStyle` property as part of specifying the alpha part for our RGBA color. For a more faded-out look, we are actually dividing the `ratio` value by two for an even smaller alpha value. If you preview your example now, you'll see our circle moving with a respectably faded-out motion trail following behind it! And with that, you are done creating a motion trail.

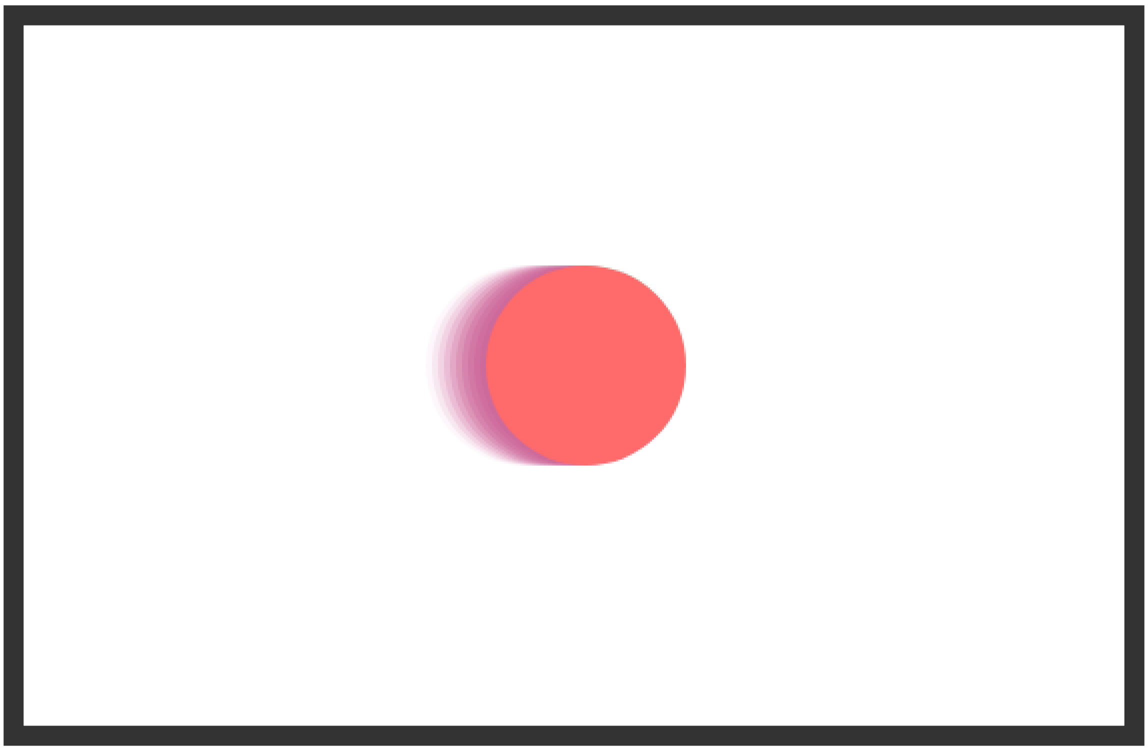
creating a motion trail.

HTML JavaScript

javascript

```
var canvas = document.querySelector("#myCanvas");
var context = canvas.getContext("2d");

var xPos = -100;
var yPos = 170;

function update() {
  context.clearRect(0, 0, canvas.width, canvas.height);

  for (var i = 0; i < positions.length; i++) {
     var ratio = (i + 1) / positions.length;

    context.beginPath();
    context.arc(positions[i].x, positions[i].y, 50, 0, 2 * Math.PI, true);
    context.fillStyle = "rgba(204, 102, 153, " + ratio / 2 + ")";
    context.fill();
  }

  context.beginPath();
  context.arc(xPos, yPos, 50, 0, 2 * Math.PI, true);
  context.fillStyle = "#FF6A6A";
  context.fill();

  storeLastPosition(xPos, yPos);

  // update position
  if (xPos > 600) {
    xPos = -100;
  }
  xPos += 3;
```

output



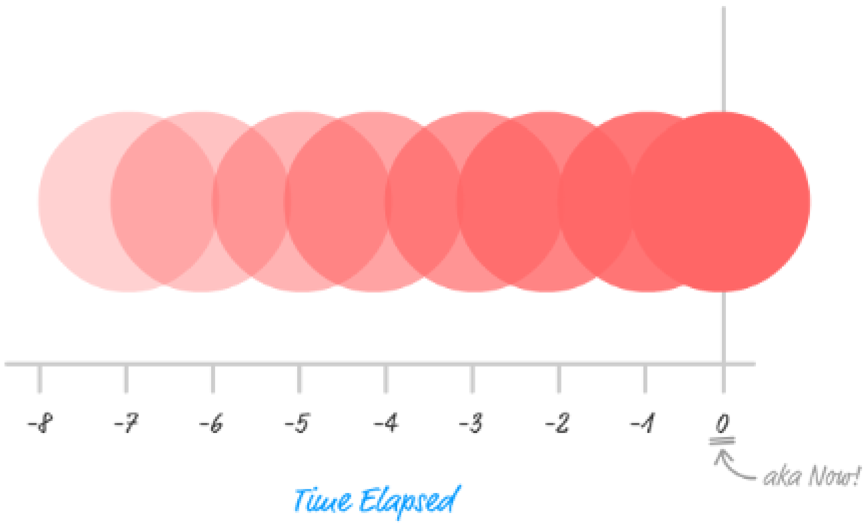
## Tying Up Some Loose Ends #

Now, before we call it a night, we talked earlier about the order in which we are doing things in. Right now, the order in our `update` function looks follows:

1. Draw the motion trail
2. Draw the source object
3. Store the source object's position

Why are we doing things in this deliberate manner? The reason has to do with something we've only casually touched upon in the past: **the canvas drawing order**. Just like painting in real life, drawing on the canvas works by layering pixels on top of older pixels.

In the life of our source object and motion trail, your source object is the shiny new thing. The end of your motion trail is where the oldest thing you are going to draw lives. We saw that with this earlier diagram:



The way our code is arranged is to allow us to respect our `canvas`'s drawing order while still ensuring we draw our source object and motion trail properly. We draw our motion trail starting with the oldest item (start of the `positions` array) and gradually moving up in time until we get to the end of our `positions` array. This allows us to layer the items in our trail properly.

The grand finale is when our source object gets drawn independently from the riff-raff that is our motion trail. Because it is the last item to get drawn, it gets top placement on the `canvas` and is drawn over the most recent motion trail item. It is at this point, we store our source object's position for the next motion trail iteration. Like clockwork, everything on our `canvas` is cleared and things start back up from the beginning.

## There is Plenty of Room for Improvement

Our motion trail implementation works, and it is the most literal translation of what we talked about towards the beginning. All of this doesn't mean that our solution can't be improved. For example, we have a lot of duplicated code between our code for drawing the source object and our code for drawing the motion trails themselves.

One optimization we can make is to move all of the drawing-related code into a `drawCircle` function that takes arguments for the position and ratio. Using that, our code looks a bit cleaner as shown in the following snippet:

```
function update() {
context.clearRect(0, 0, canvas.width, canvas.height);

 for (var i = 0; i < positions.length; i++) {
   var ratio = (i + 1) / positions.length;
   drawCircle(positions[i].x, positions[i].y, ratio);
 }

 drawCircle(xPos, yPos, "source");

 storeLastPosition(xPos, yPos);

 // update position
 if (xPos > 600) {
   xPos = -100;
 }
 xPos += 3;

 requestAnimationFrame(update);
}
update();

function drawCircle(x, y, r) {
 if (r == "source") {
   r = 1;
```

```
  } else {
    r /= 4;

  }

  context.beginPath();
  context.arc(x, y, 50, 0, 2 * Math.PI, true);
  context.fillStyle = "rgba(204, 102, 153, " + r + ")";
  context.fill();
}
```

This code should contain no surprises...mostly. The only strange thing we are doing is passing in a value of **source** as opposed to a numerical ratio when our source object is being drawn. This ensures that our source object is always drawn with an opacity of 1. For all motion trail-related drawing, the usual ratio values are used.

This is just one example of the sort of optimization you can make. You have a lot of runway when implementing motion trails, so go crazy!