Lab Activity 2

In this lab activity we are going to focus on improving the performance of our webGL applications. Head to the below CodeSandbox where you will find a very simplified version of our painting assignment:

https://codesandbox.io/s/lab-2-template-3jy62

You'll notice that the interface allows only for us to draw circles by clicking on the **Add Shapes** button, along with an FPS meter in the top right which we will get to later:

What We Did Last Time

In the painting assignment we focused on the process of getting things to render on screen. In doing so, we broke many '*best practices*'. This was to keep things a bit more clear as to what the program was actually doing. Now that you have a slightly better understanding of how webGL works under the hood, we should note these '*best practices*'.

Reduce, Reuse, Recycle

As you may have heard us mention before, one of the biggest pitfalls in rendering is re-creating objects in loops (or anywhere it isn't necessary, really). This includes but is not limited to:

- `Float32Array` 's
- Buffers created for use on the GPU using `gl.createBuffer()`

In our painting assignment, if you followed along exactly, then you will have fallen into these pitfalls. Consider the second half of the render function in the `Circle` class which you can find in the provided `Circle.js` file:

```
render(gl) {
      . . .
    // Create a buffer object
    var vertexBuffer = gl.createBuffer();
    if (!vertexBuffer) {
      console.log("Failed to create the buffer object");
      return -1;
    }
    // Bind the buffer object to target
    gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
    // Write date into the buffer object
    gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices),
          gl.DYNAMIC_DRAW);
    // Assign the buffer object to aPosition variable and enable
    gl.vertexAttribPointer(aPosition, 2, gl.FLOAT, false, 0, 0);
    gl.enableVertexAttribArray(aPosition);
    gl.drawArrays(gl.TRIANGLES, 0, 3);
  }
}
```

Notice we do both of the things we were warned against doing earlier. In every single shape, every time it renders it recreates itself almost entirely! So how do we fix things?

The Render Loop

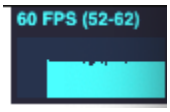
In order to properly visualize how these operations affect our program's performance we will want to measure the time it takes to draw our scene, not just when we add a new shape, but over a sufficiently long period.

So, let's enable the render loop.

> **NOTE:** The render loop will continuously render our scene on every frame. Each computer differs in performance, but the maximum frames per second we can get in a browser with default settings is 60fps. The meter in the CodeSandbox will measure the time it takes to render our scene and how many times it manages to render over the course of one second. It will also show us the running average of each frame's render time. This will give us a more accurate representation of our performance as we get closer to the real-world performance of a web application.

Take a look at line 98 in our `index.js` file, and uncomment it. Your FPS meter should now be collecting render data:

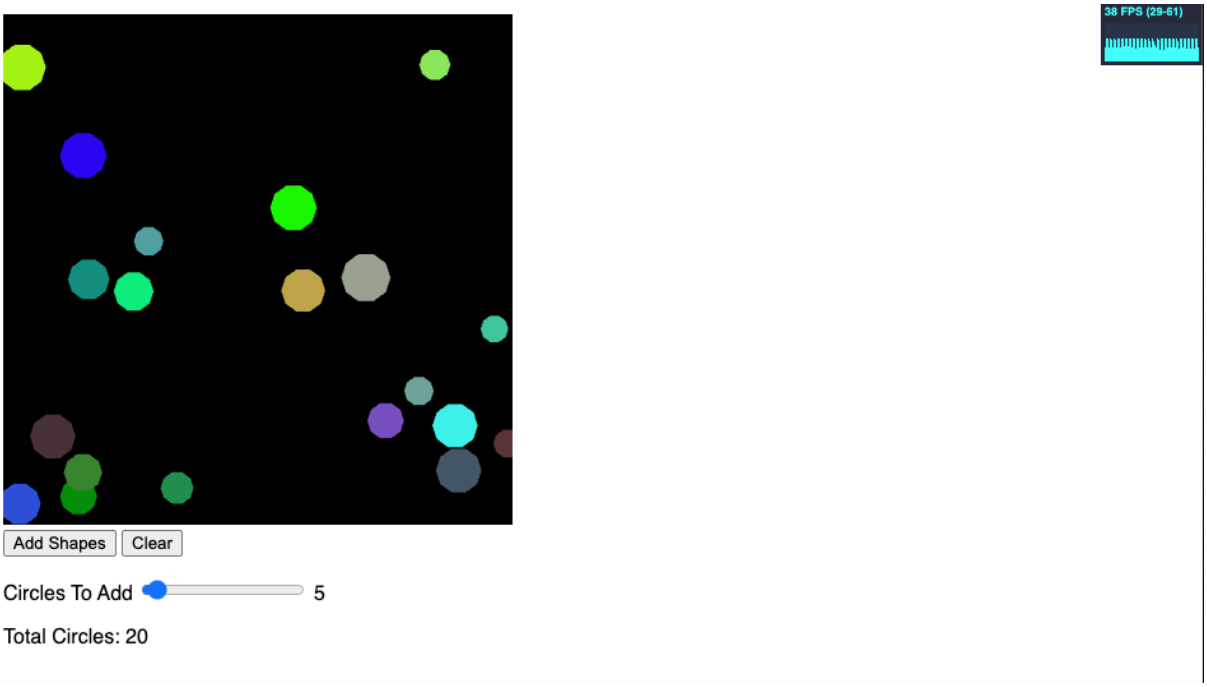
```
97  // start render loop
98  tick();
```



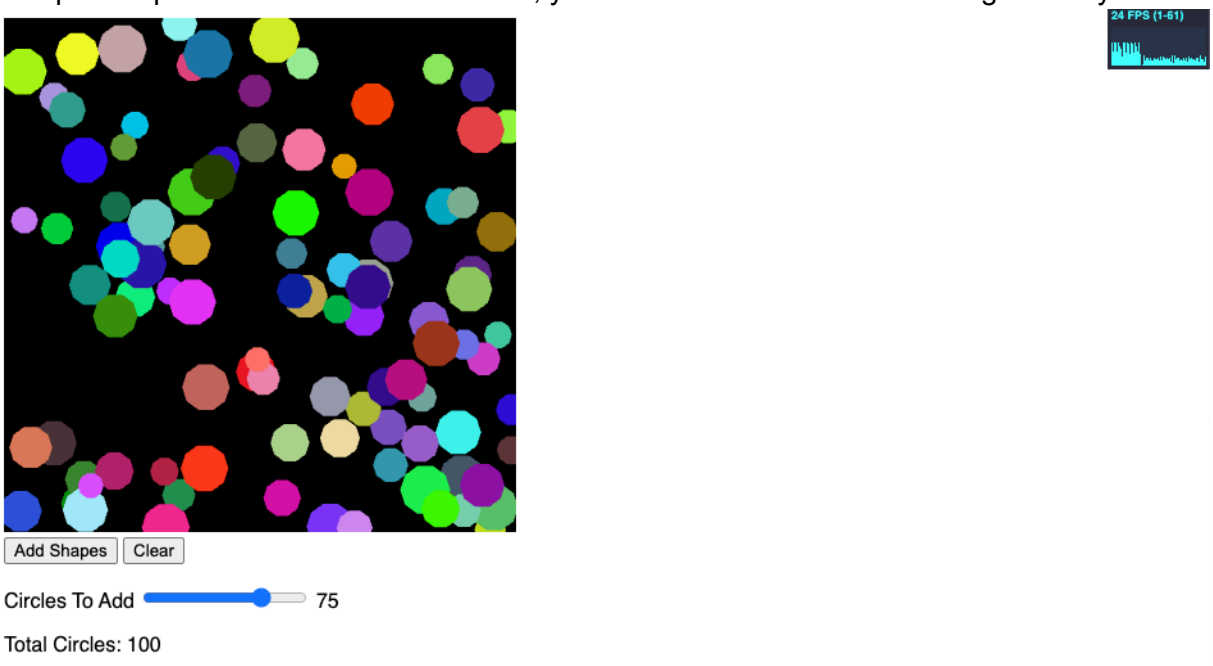
You can examine the `tick()` function if you want to see how the render data is collected. It essentially marks the time before and after each render frame and keeps that running average under the hood.

Create Some Expensive Circles

Now that we have our render loop running, let's create some circles. Click the **Add Shapes** button until you have 25 circles. Depending on your computer, you may begin to see some drops in performance:



Bump that up to 100 circles. Chances are, your browser has slowed down significantly:



Hit the clear button before you set fire to your computer :)

What should we do to increase performance here, then? Well, if the problem is that we are recreating buffers and array objects, then let's just re-use the ones we already made!

Reusing Our Allocated Objects

Now consider the following lines from our `Circle.render()` function:

```
// Create a buffer object
    var vertexBuffer = gl.createBuffer();
    if (!vertexBuffer) {
      console.log("Failed to create the buffer object");
      return -1;
    }
    . . .
    // Write date into the buffer object
    gl.bufferData(
      gl.ARRAY_BUFFER,
      new Float32Array(vertices),
      gl.DYNAMIC_DRAW
    );
```

How can we store the buffer and vertex array such that we can re-use it on each following render? Well, since this is a class we can simply create a reference to it in the constructor:

```
constructor() {
   this.type = "circle";
   this.position = [0.0, 0.0, 0.0];
   this.color = [1.0, 1.0, 1.0, 1.0];
   this.size = 5.0;
   this.segments = 10;

   // add this, will update on first render
   this.buffer = null;
}
```

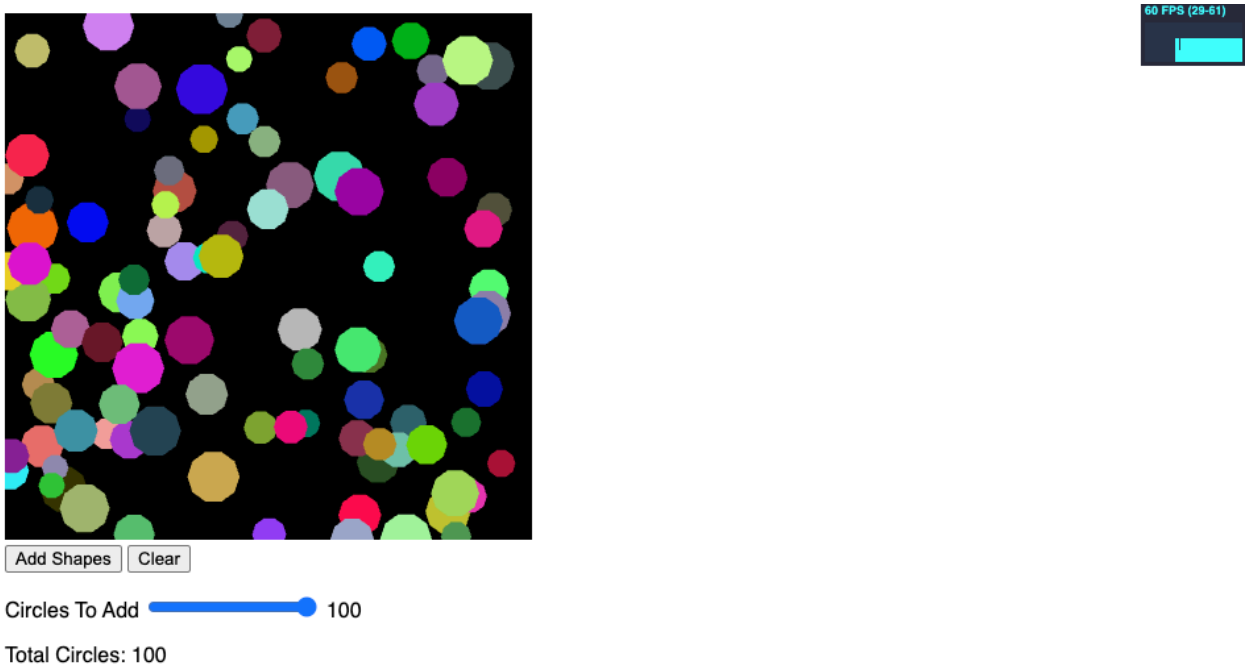
And then in the render function, change this:

```
var vertexBuffer = gl.createBuffer();
if (!vertexBuffer) {
    console.log("Failed to create the buffer object");
    return -1;
}
```

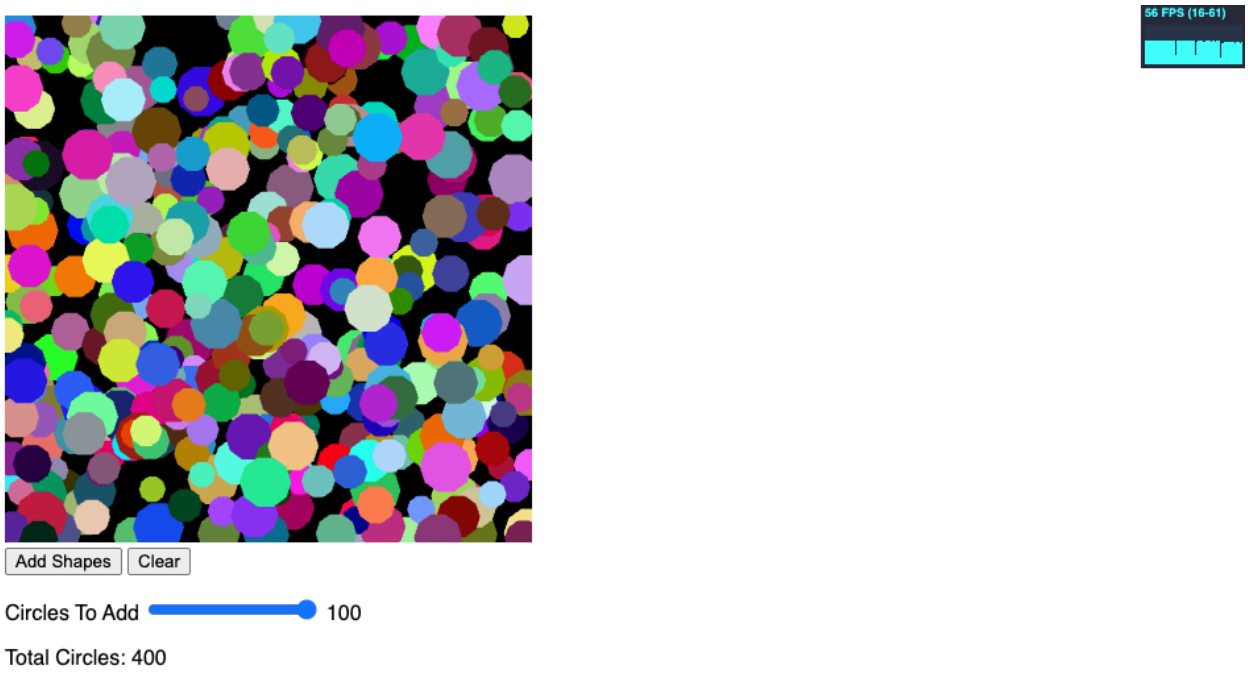
To this:

```
if (this.buffer === null) {
    // Create a buffer object
    this.buffer = gl.createBuffer();
    if (!this.buffer) {
        console.log("Failed to create the buffer object");
         return -1;
    }
}
```

Now let's put 100 circles on screen again, you should be sitting pretty on a nice framerate:



Keep adding circles until it starts to slow down, and take note of the total. I start to see a small decline (~4fps) once I hit 400:



And even with 1000 circles, I have the same FPS as I did with only 100 circles earlier! That's literally a 10x improvement with a couple lines of code.

Well let's go even further. Remember how we were recreating the vertex array in the render function as well? Let's change that too. It's important to note what is happening in the render function here. We are generating triangles to create our circles out of and then drawing each triangle in a loop.

What we will want to do instead is to generate these vertices *once* and store them to be rendered later. Let's move all the logic that generates the vertices into its own function and add a reference to the constructor:

```
class Circle {
 constructor() {
   this.buffer = null;
   this.vertices = null; // add this
 }

  generateVertices() { // add function
    // move logic here
  }
  . . .
}
```

The easiest way to do this is to steal the code from the render function, first grab this stuff **(be very careful that you grab only the below stuff, note the dots showing that there is code between what we are grabbing)**:

```
let [x, y] = this.position;
. . .
var d = this.size / 200.0; // delta

let angleStep = 360 / this.segments;
for (var angle = 0; angle < 360; angle = angle + angleStep) {
  let centerPt = [x, y];
  let angle1 = angle;
  let angle2 = angle + angleStep;
  let vec1 = [
    Math.cos((angle1 * Math.PI) / 180) * d,
    Math.sin((angle1 * Math.PI) / 180) * d
  ];
  let vec2 = [
    Math.cos((angle2 * Math.PI) / 180) * d,
    Math.sin((angle2 * Math.PI) / 180) * d
  ];
  let pt1 = [centerPt[0] + vec1[0], centerPt[1] + vec1[1]];
  let pt2 = [centerPt[0] + vec2[0], centerPt[1] + vec2[1]];
  . . .
  // leave everything else, but make sure its not in a for loop anymore
}
```

Make sure everything else below that code remains, but note it is no longer in a for loop. So we will need to do some clean up later.

Move that code into the function; we also need to create an array to hold the generated vertices. In the for loop, instead of drawing each triangle one at a time we are going to store them all in a single array. Note the two added lines below:

```
generateVertices() {
  let [x, y] = this.position;
  var d = this.size / 200.0;

  let v = []; // ADD THIS
  let angleStep = 360 / this.segments;
  for (var angle = 0; angle < 360; angle = angle + angleStep) {
    let centerPt = [x, y];
    let angle1 = angle;
    let angle2 = angle + angleStep;
    let vec1 = [
      Math.cos((angle1 * Math.PI) / 180) * d,
      Math.sin((angle1 * Math.PI) / 180) * d
    ];
    let vec2 = [
      Math.cos((angle2 * Math.PI) / 180) * d,
      Math.sin((angle2 * Math.PI) / 180) * d
    ];
    let pt1 = [centerPt[0] + vec1[0], centerPt[1] + vec1[1]];
    let pt2 = [centerPt[0] + vec2[0], centerPt[1] + vec2[1]];

    v.push(x, y, pt1[0], pt1[1], pt2[0], pt2[1]); // ADD THIS
  }
}
```

Then update the reference in our class:

```
generateVertices() {
    . . .
    v.push(x, y, pt1[0], pt1[1], pt2[0], pt2[1]);
  }
   this.vertices = new Float32Array(v); // add this
}
```

Now we need to clean up the `render()` function like we mentioned before. Firstly, we need to make sure the vertices are generated on the first render. Then we need to make sure the `bufferData()` call gets our stored vertices instead of creating a new array. Finally, the draw call is currently drawing 3 vertices explicitly. Since we have all of our circles' vertices available right away, we can draw the whole thing in one call! Note that the stored vertex array contains 2 elements for each vertex, x and y, so it's length is two times the amount of vertices we have.

To do all of this (pay attention to the comments!):

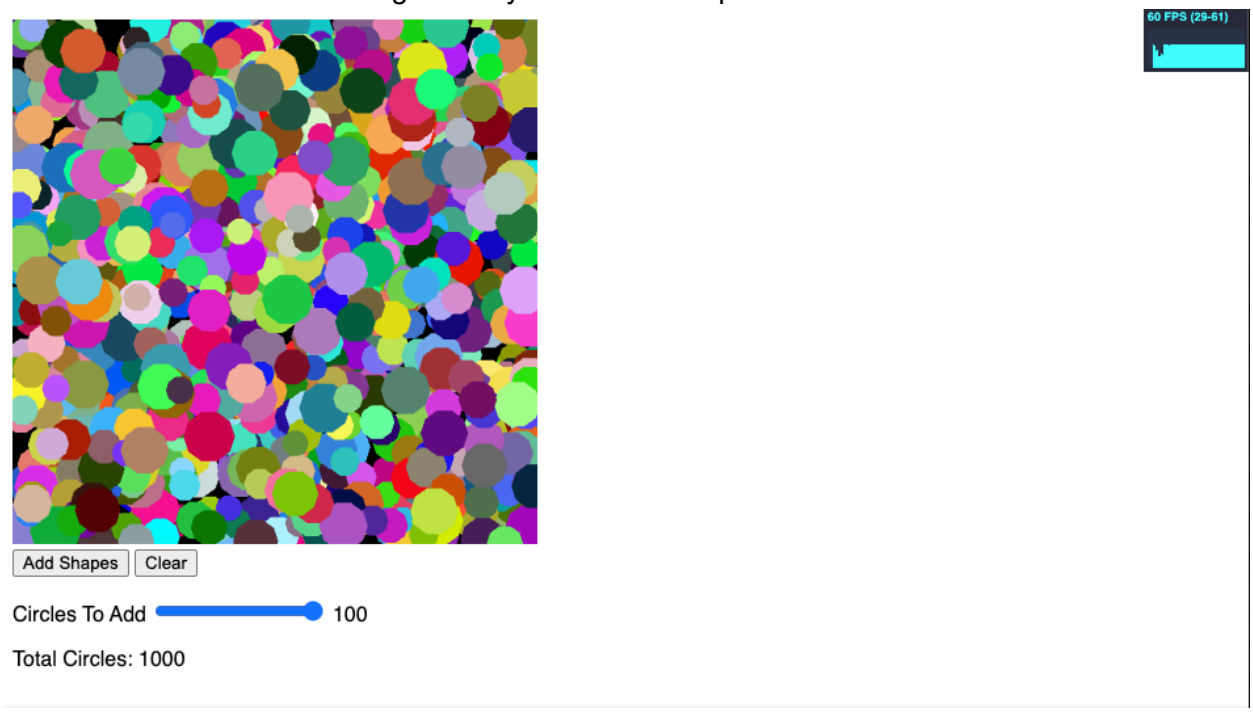
```
render(gl) {
  . . .
  // add next 3 lines
  if (this.vertices === null) {
    this.generateVertices();
  }

  // creating the buffer is the same as before
  if (this.buffer === null) {
    this.buffer = gl.createBuffer();
    if (!this.buffer) {
      console.log("Failed to create the buffer object");
      return -1;
    }
  }
  . . .
  // note the second parameter here has changed
  gl.bufferData(gl.ARRAY_BUFFER, this.vertices, gl.DYNAMIC_DRAW);
  . . .
  // note the third parameter here has changed
  gl.drawArrays(gl.TRIANGLES, 0, this.vertices.length / 2);
}
```

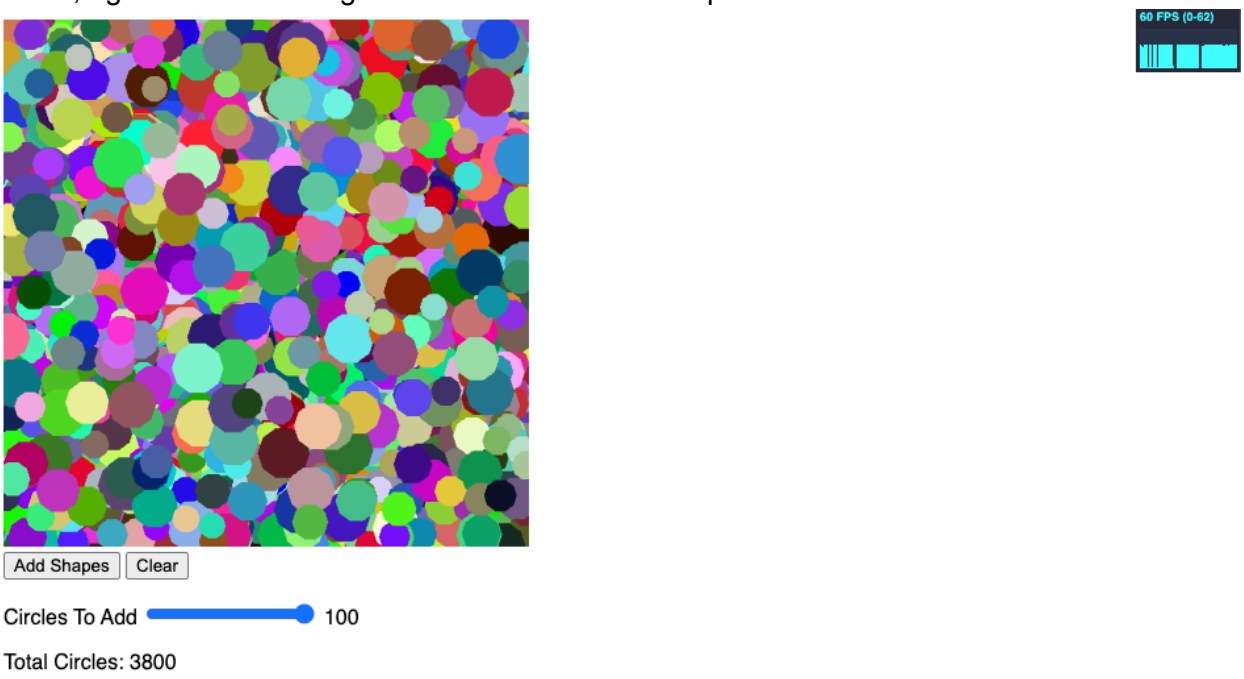
To make sure things are not all jumbled up, your render function should now do the following, in this order:

- Grab the color of the circle
- Grab the uniform and attribute positions
- Set the uniform uFragColor
- Check if vertices are generated
    - If not, generate them
- Check for buffer
    - If not, create it
- Bind the buffer (`bindBuffer()`)
- Write the data to the buffer (`bufferData()`)
- Assign buffer to attribute (`vertexAttribPointer()`)
- Enable the attribute array
- Draw

Now let's check our performance! Pop 1000 circles on the screen. After the circles are drawn (it can take a second to generate them the first time), you should be looking at a ***much*** better framerate. With 1000 circles I get a very consistent 60 fps:



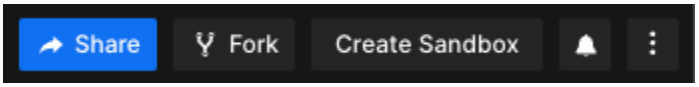
In fact, I got tired of creating more before it started to dip:



How many circles can you get on screen before it starts to slow down?

Submission

To submit this lab assignment, make sure you submit your ***own*** codesandbox link for the lab. This means that you will need to fork the sandbox that you and your group worked on together such that each individual has their own. The fork button is on the top right corner of the page:



Then, you will submit the following on canvas:

- Link to your sandbox (**click on the blue share button and click copy link at bottom**)
- List of groupmates names