# Boomshine-G(ood thing there's even more)

We're not going to require you to do any more gameplay on Boomshine (besides Boomshine F), but we are going to use it to cover some more things you may need for Project 2.

Make a copy of the **Boomshine-E folder** and name it **Boomshine-G**

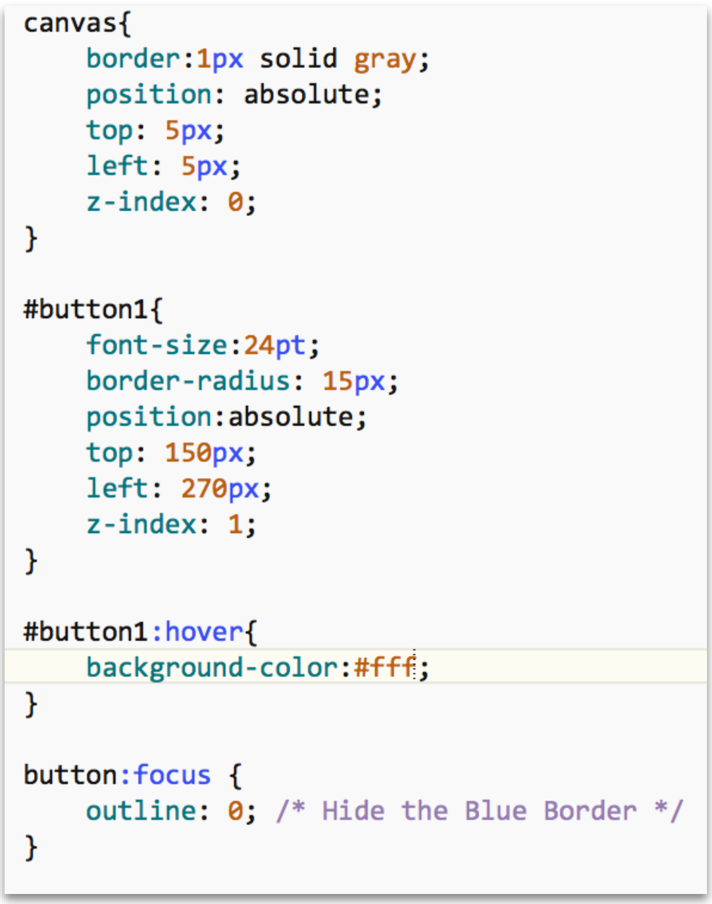
## Part I - Buttons

1) First we'll look at how to add a button to our Boomshine project.

A) Here's the HTML:

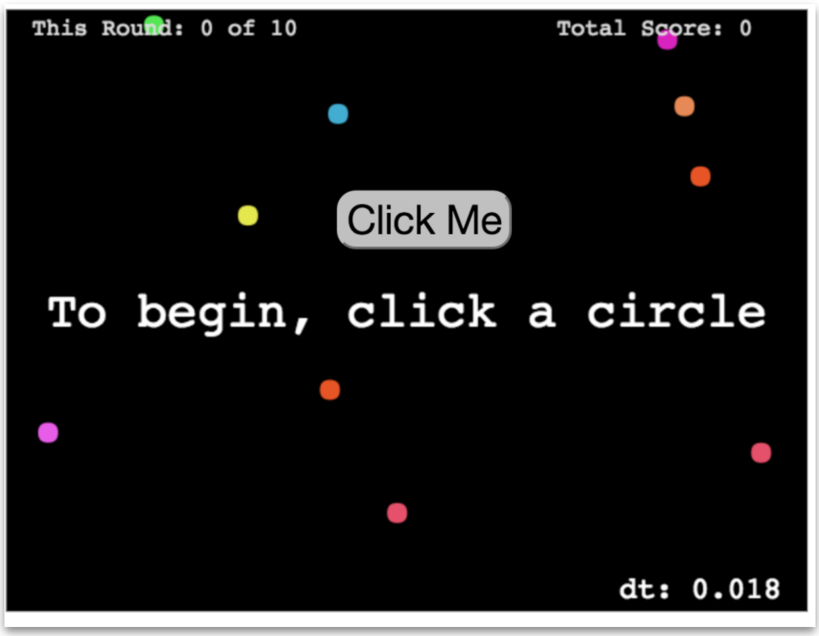
```
<button id="button1">Click Me</button>
```

B) Here's the CSS - recall you'll have to absolutely position the `<button>` and the `<canvas>`. (We're not worried about the `<audio>` elements because they are hidden anyway)



```
canvas{
    border:1px solid gray;
    position: absolute;
    top: 5px;
    left: 5px;
    z-index: 0;
}

#button1{
    font-size:24pt;
    border-radius: 15px;
    position:absolute;
    top: 150px;
    left: 270px;
    z-index: 1;
}

#button1:hover{
    background-color:#fff;
}

button:focus {
    outline: 0; /* Hide the Blue Border */
}
```

C) Reload the page - the button should be visible - you also get “mouse down” and “mouse over” states for free:



D) You can hook up event listeners to this button as usual. Obviously, you only show this button when the app is in the proper state. You can hide and show it in JavaScript with:

```
document.querySelector("#button1").style.display = "none";
```

and

```
document.querySelector("#button1").style.display = "inline";
```

If you find the standard HTML `<button>` boring, you can always create a cool button graphic and use CSS to make that the background of the button.

2) You could also draw "buttons" directly onto the canvas, and then check to see if the user clicked in the containing rectangle of the button in `doMousedown()`.

A) Here's a helper method you can add to **utilities.js**

```
// http://24bitjs.com/2014-11-how-to-detect-if-a-point-is-inside-a-rectangle-in-javascript/

function rectangleContainsPoint(rect, point) {
	// example
	// rect = {x:0,y:0,width:10,height:10}
	// point = {x:5,y:5}
	// would return true
	if (rect.width <= 0 || rect.height <= 0) {
		return false;
	}
	return (point.x >= rect.x && point.x <= rect.x + rect.width &&
point.y >= rect.y && point.y <= rect.y + rect.height);
};
```

B) We then call it in `doMouseDown()`:

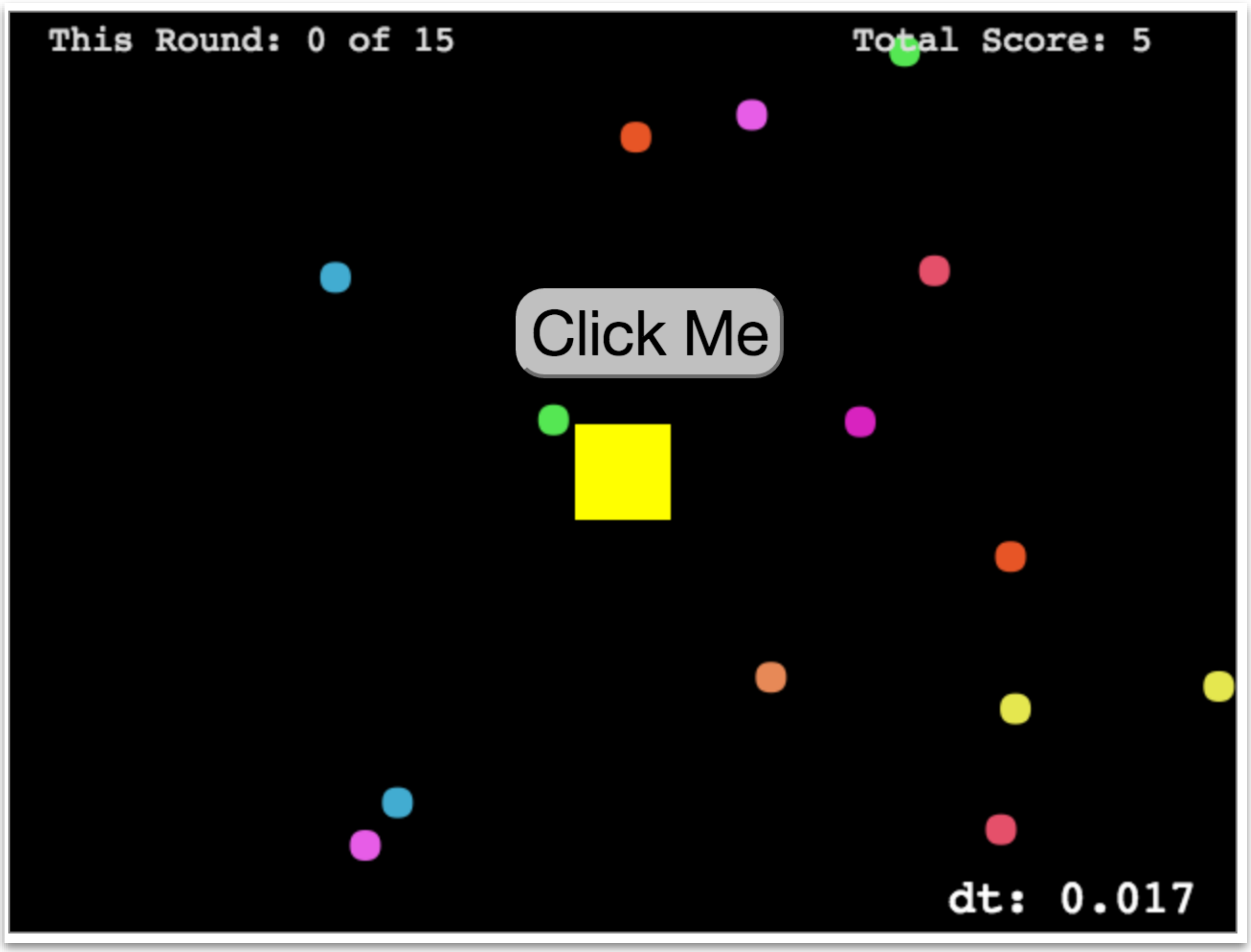
```
var mouse = getMouse(e);
var rect = {x:295,y:215,width:50,height:50};

if(this.gameState == this.GAME_STATE.ROUND_OVER &&
rectangleContainsPoint(rect,mouse)){
	this.gameState = this.GAME_STATE.DEFAULT;
	this.reset();
	return;
}
```

C) We can draw the button in `drawHUD()`:

```
ctx.fillStyle = "yellow";
ctx.fillRect(295,215,50,50);
```

D) Reload the page. Whenever we're at the "round over" state, we can only advance by clicking in the yellow box.

But … we don't have a rollover state or a down state for our canvas button, we'll have to draw those ourselves in `update()`. See why absolutely positioned `<button>` elements are easier?



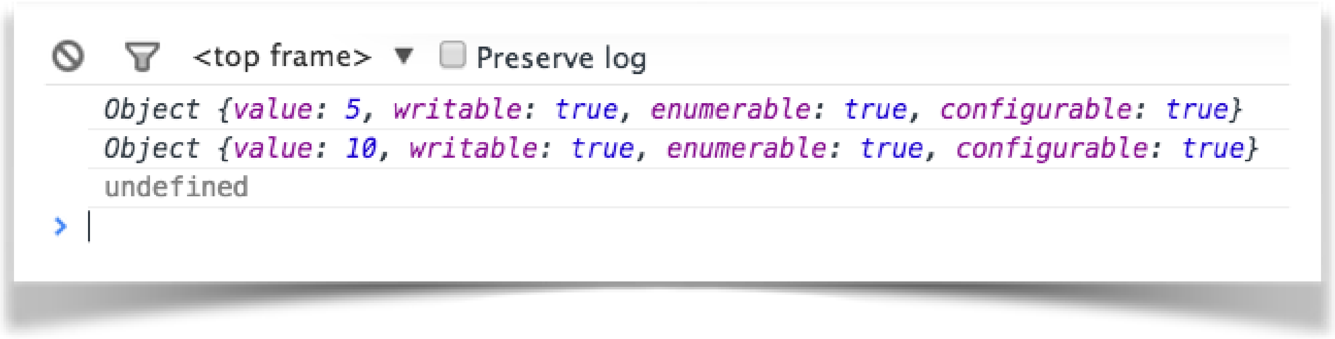
## Part II - A little more with the ECMAScript5 Object() methods

These Object methods let you control characteristics of object properties.

A) `Object.getOwnPropertyDescriptor(object, 'propertyName')`

gives the characteristics of that object's properties.

```
“use strict”;
var ship = {xSpeed: 5, ySpeed : 10};
console.log(Object.getOwnPropertyDescriptor(ship, ‘xSpeed'));
console.log(Object.getOwnPropertyDescriptor(ship, ‘ySpeed'));
console.log(Object.getOwnPropertyDescriptor(ship, 'boo'));
```



```
Object {value: 5, writable: true, enumerable: true, configurable: true}
Object {value: 10, writable: true, enumerable: true, configurable: true}
undefined
>
```

**Property (or data) descriptions:**
value: duh
writable: we can change the value
enumerable: the property shows up in a for..in loop
configurable: we can change the property descriptor definition - example:

```
Object.defineProperty(ship, 'xSpeed',{
        value: 500,             // let's change the value
        writable: false,        // let's make it read only
        configurable: false,    // not configurable anymore!
        enumerable: true        // can be used in an iterator
});
console.log(ship); // ship.xSpeed = 500
ship.xSpeed = 100; // fails! xSpeed is now read only
```

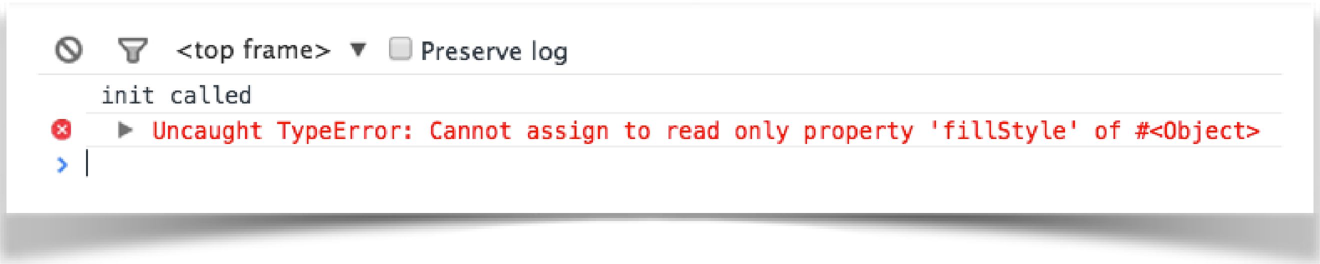
Where is this useful? We could make the `fillStyle` property of a circle **read-only** by adding a line of code like this to the for loop of makeCircles():

```
Object.defineProperty(c,'fillStyle',{writable: false});
```

and then if later in drawCircles() we tried this:

```
c.fillStyle  ="purple";
```

We'd get this:



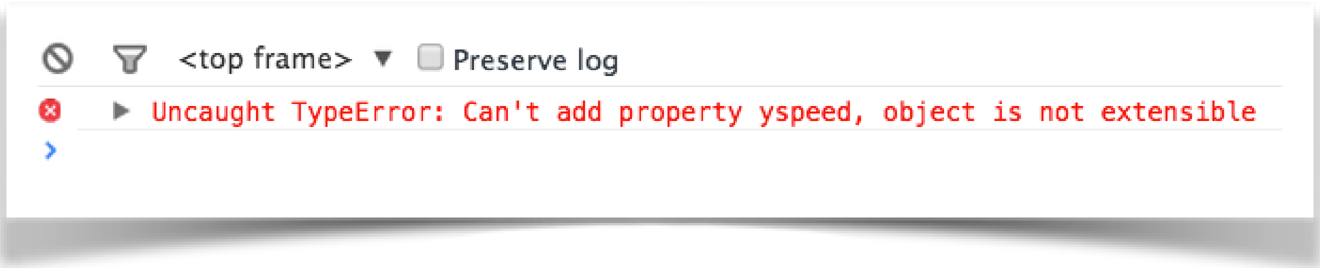
```
<top frame> ▼ Preserve log
init called
▶ Uncaught TypeError: Cannot assign to read only property 'fillStyle' of #<Object>
> |
```

B) Object.preventExtensions() - this prevents new properties from being added to an object. Where would this come in handy? Imagine this:

```
“use strict”;
var ship = {xSpeed: 5, ySpeed : 10};
// much later
ship.xSpeed = 6; // OK
ship.yspeed = 6; // logic error - we misspelled ySpeed and added
// a new property to ship
```

add this right after we create the ship:
Object.preventExtensions(ship);

and we'll get a run time error,  which is easier to track down than a logic error:



```
<top frame> ▼ Preserve log
▶ Uncaught TypeError: Can't add property yspeed, object is not extensible
>
```

Where can you use this in Boomshine?
Add:

```
Object.preventExtensions(c);
```

to the end of the for loop in makeCircles().

Then if you accidentally misspelled a variables later on (ex. in moveCircles(), misspelling c.lifetime as c.liiftime), you would immediately get a run time error, rather than that hard to track down logic error.

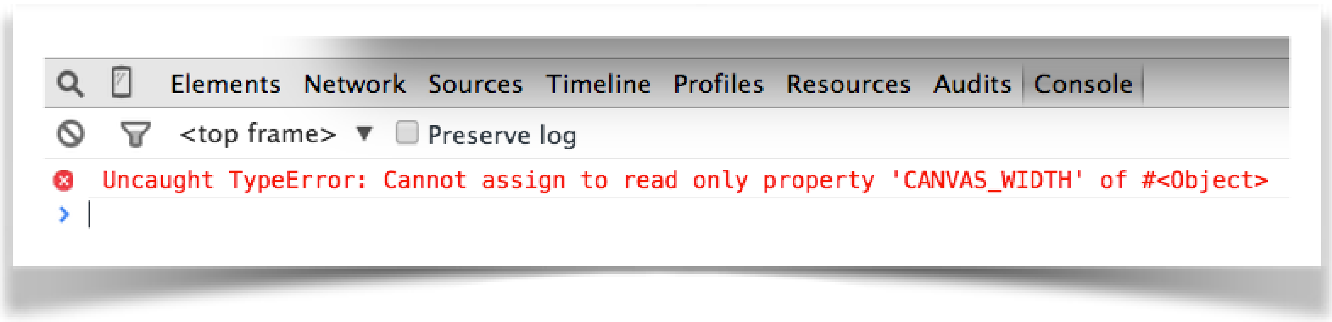
C) Object.seal() - similar to the above, it takes an object, calls preventExtensions() on it, and then sets all of the properties to configurable:false. The values of properties can still be changed.

Where can you use this in Boomshine?
Change: Object.preventExtensions(c); above to Object.seal(c);

D) Object.freeze() - the highest level of data protection, it essentially calls Object.seal() on an object, and then sets all of the properties of that object to writable:false.

Where can you use this in Boomshine? Create a single Game object with all of your constants as properties:

```
var Game = Object.freeze({
	WIDTH:640,
	HEIGHT:480,
	…
});

Game.WIDTH = 1000; // Error!
```



```
Uncaught TypeError: Cannot assign to read only property 'CANVAS_WIDTH' of #<Object>
```

**Reference:**
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getOwnPropertyDescriptor

https://github.com/getify/You-Dont-Know-JS/blob/master/this%20&%20object%20prototypes/ch3.md

## Part III - Turning myKeys into a module

Let's move the `myKeys` array out of the global scope and make it a module. This will help to clean up some of our code over in app.main too.

1) Make the top of **keys.js** look like this:

```
"use strict";

var app = app || {};

// start IIFE
app.myKeys = function(){
      var myKeys = {};
      …
      …
```

2) Make the bottom of **keys.js** look like this:

```
    …
    …
    return myKeys;
}() // end IIFE
```

3) To update the sandbox code and make `myKeys` a property of main, add this line to `window.onload` (in **loader.js**), right before `app.main.init()`:

```
app.main.myKeys = app.myKeys;
```

4) Add a new property to `app.main`:

```
myKeys: undefined,   // required - loaded by main.js
```

5) In `app.main`, replace `myKeys` with `this.myKeys` (this should be in 4 places)

6) Test your "Shift up-arrow" cheat to make sure they keyboard polling still works.

## Part IV - Function Constructors

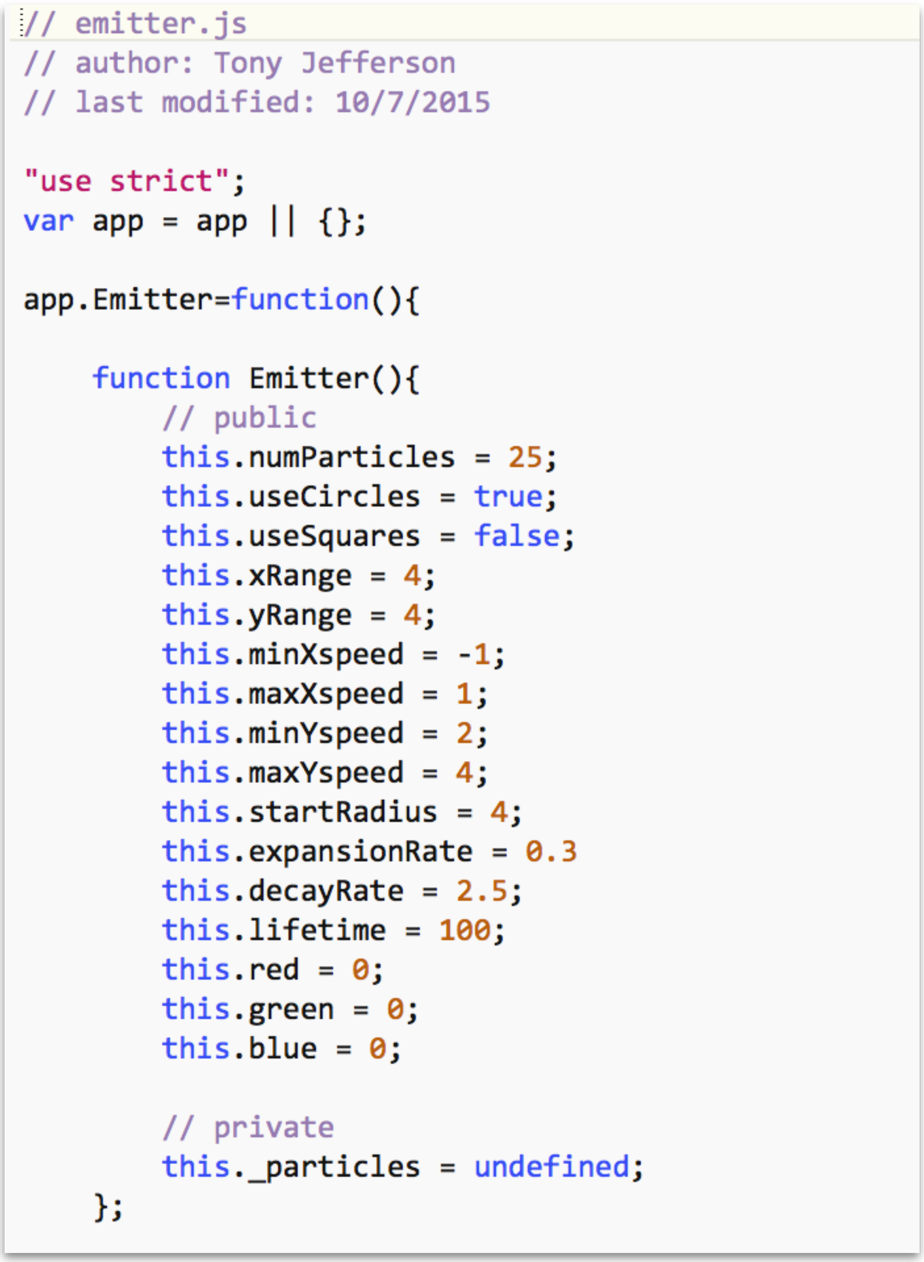
If you haven't yet, go read the Function Constructor handout.

## Part V - Adding a Particle System module to Boomshine

We have created a particle system within a *Function Constructor* called **Emitter**. The source code is up in mycourses - and here it is below:

Constructor —>

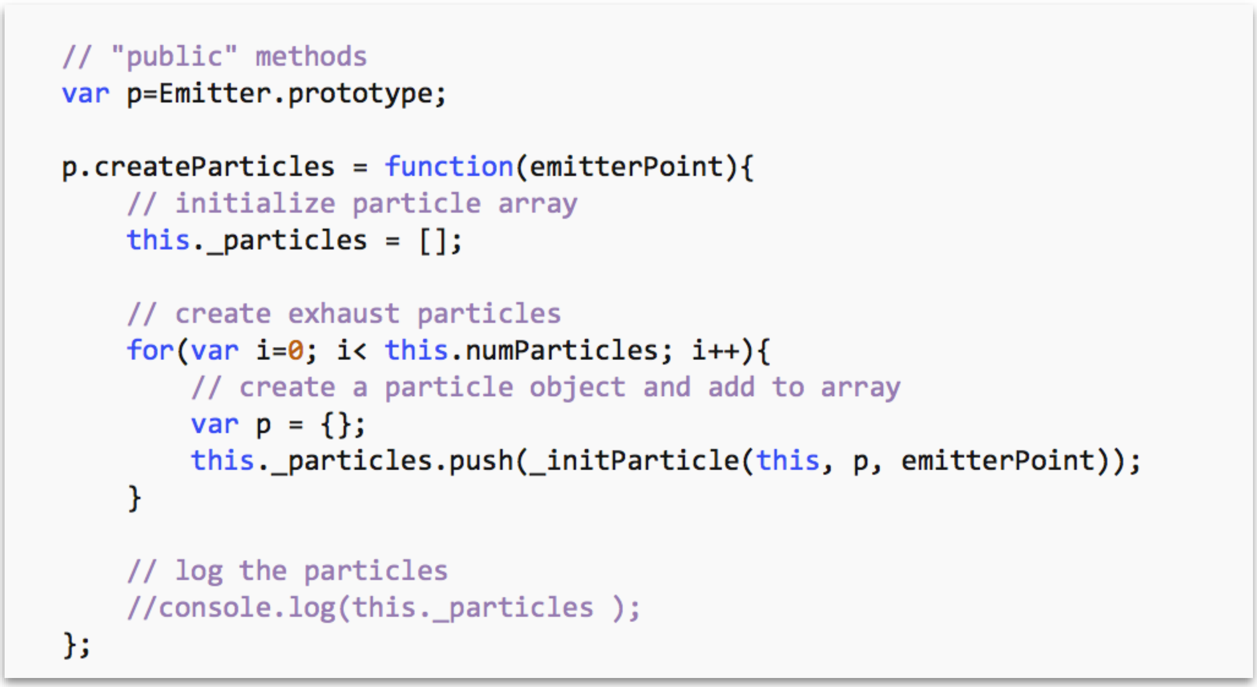
Note that "private" properties (those we don't want the users of this object to directly access) begin with an underscore - this is a common convention.



```
// emitter.js
// author: Tony Jefferson
// last modified: 10/7/2015

"use strict";
var app = app || {};

app.Emitter=function(){

    function Emitter(){
        // public
        this.numParticles = 25;
        this.useCircles = true;
        this.useSquares = false;
        this.xRange = 4;
        this.yRange = 4;
        this.minXspeed = -1;
        this.maxXspeed = 1;
        this.minYspeed = 2;
        this.maxYspeed = 4;
        this.startRadius = 4;
        this.expansionRate = 0.3
        this.decayRate = 2.5;
        this.lifetime = 100;
        this.red = 0;
        this.green = 0;
        this.blue = 0;

        // private
        this._particles = undefined;
    };
```

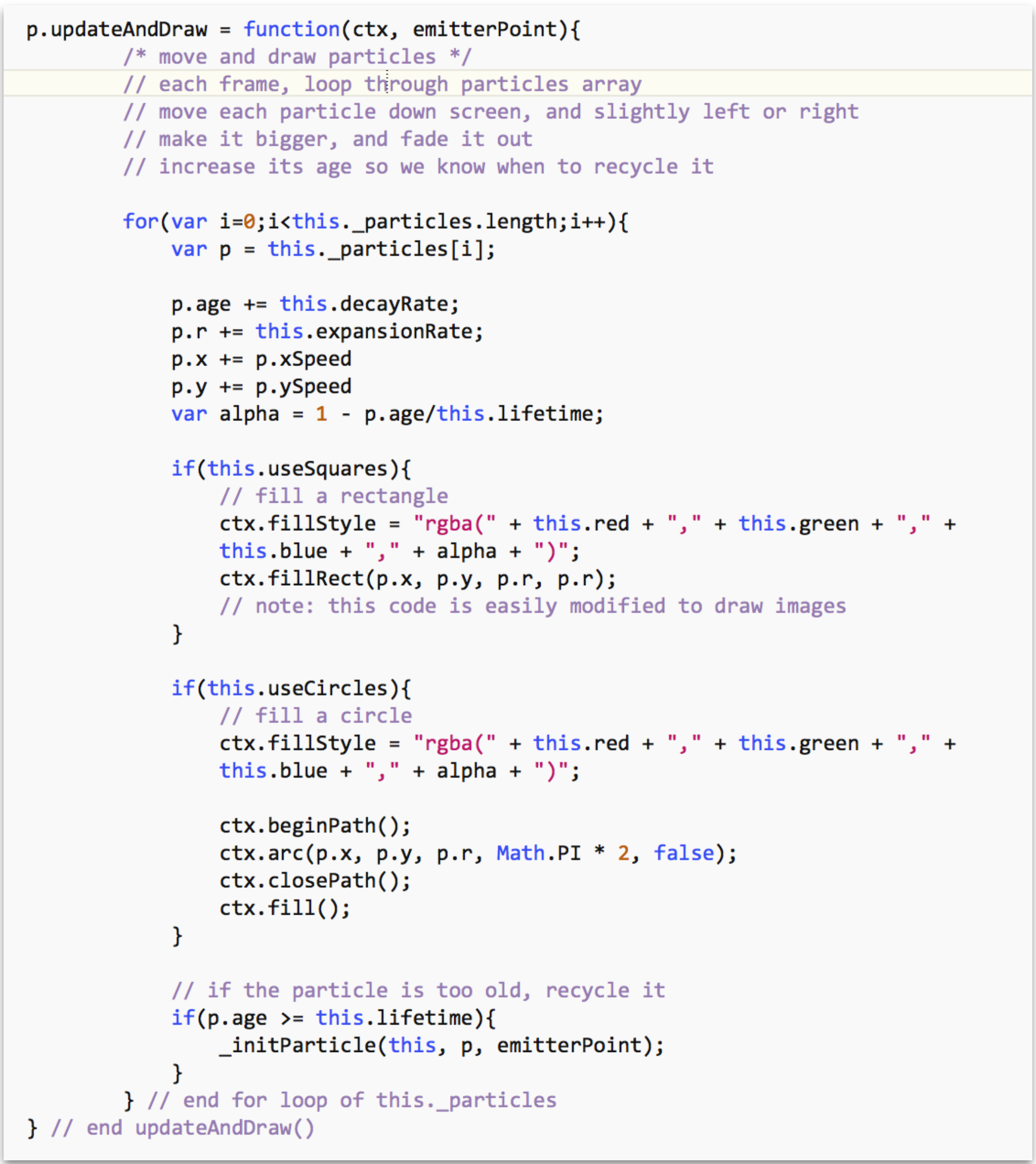
Here we attach a `createParticles()` method to the `.prototype` - which makes it available to every instance of **Emitter**.

PS - `var p=Emitter.prototype;` is just a handy alias (shortcut).



```
// "public" methods
var p=Emitter.prototype;

p.createParticles = function(emitterPoint){
    // initialize particle array
    this._particles = [];

    // create exhaust particles
    for(var i=0; i< this.numParticles; i++){
        // create a particle object and add to array
        var p = {};
        this._particles.push(_initParticle(this, p, emitterPoint));
    }

    // log the particles
    //console.log(this._particles );
};
```

The typical `update()` and `draw()` methods are combined into `updateAndDraw()` as an optimization. Note that we're not using `dt` here - we're optimistically assuming 60 FPS - but we probably should make it a parameter of this function and divide all of the relevant property values by 60. (… but I'll leave that up to you to do at some point …)



```
p.updateAndDraw = function(ctx, emitterPoint){
        /* move and draw particles */
        // each frame, loop through particles array
        // move each particle down screen, and slightly left or right
        // make it bigger, and fade it out
        // increase its age so we know when to recycle it

        for(var i=0;i<this._particles.length;i++){
            var p = this._particles[i];

            p.age += this.decayRate;
            p.r += this.expansionRate;
            p.x += p.xSpeed
            p.y += p.ySpeed
            var alpha = 1 - p.age/this.lifetime;

            if(this.useSquares){
                // fill a rectangle
                ctx.fillStyle = "rgba(" + this.red + "," + this.green + "," +
                this.blue + "," + alpha + ")";
                ctx.fillRect(p.x, p.y, p.r, p.r);
                // note: this code is easily modified to draw images
            }

            if(this.useCircles){
                // fill a circle
                ctx.fillStyle = "rgba(" + this.red + "," + this.green + "," +
                this.blue + "," + alpha + ")";

                ctx.beginPath();
                ctx.arc(p.x, p.y, p.r, Math.PI * 2, false);
                ctx.closePath();
                ctx.fill();
            }

            // if the particle is too old, recycle it
            if(p.age >= this.lifetime){
                _initParticle(this, p, emitterPoint);
            }
        } // end for loop of this._particles
} // end updateAndDraw()
```

This method makes 1 particle and is intended to be called only by **Emitter** itself. We pass this method a particle as the first argument and thus "recycle" particles - we never have to make any

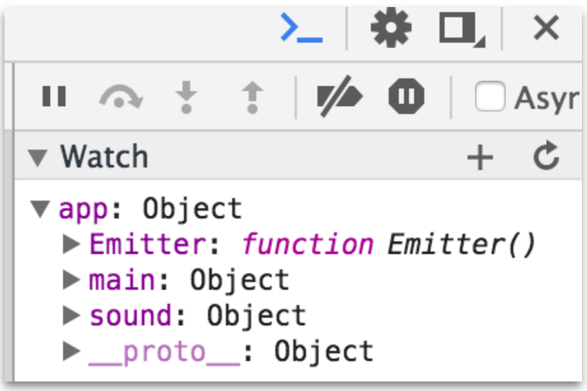
```
    // "private" method
    function _initParticle(obj, p, emitterPoint){

        // give it a random age when first created
        p.age = getRandom(0,obj.lifetime);

        p.x = emitterPoint.x + getRandom(-obj.xRange, obj.xRange);
        p.y = emitterPoint.y + getRandom(0, obj.yRange);
        p.r = getRandom(obj.startRadius/2, obj.startRadius); // radius
        p.xSpeed = getRandom(obj.minXspeed, obj.maxXspeed);
        p.ySpeed = getRandom(obj.minYspeed, obj.maxYspeed);
        return p;
    };


    return Emitter;
}();
```

more particles that were initially created.

1) Download **emitter.js** from mycourses and put it into your **js** folder.

2) Add a `<script>` tag to **boomshine.html**

3) Reload the page and check the debugger, app should now have an `Emitter` property - I'm "watching" the `app` global in the debugger.



```
▼ app: Object
  ▶ Emitter: function Emitter()
  ▶ main: Object
  ▶ sound: Object
  ▶ __proto__: Object
```

4) In **loader.js,** in the onload block, add this code before `app.main.init()`:

```
app.main.Emitter = app.Emitter;
```

5) Add this property to `app.main`:

```
Emitter : undefined,  // required - loaded by main.js
```

6) Now we'll write code to test the Emitter. Add these properties to `app.main`:

```
pulsar : undefined,
exhaust : undefined,
```

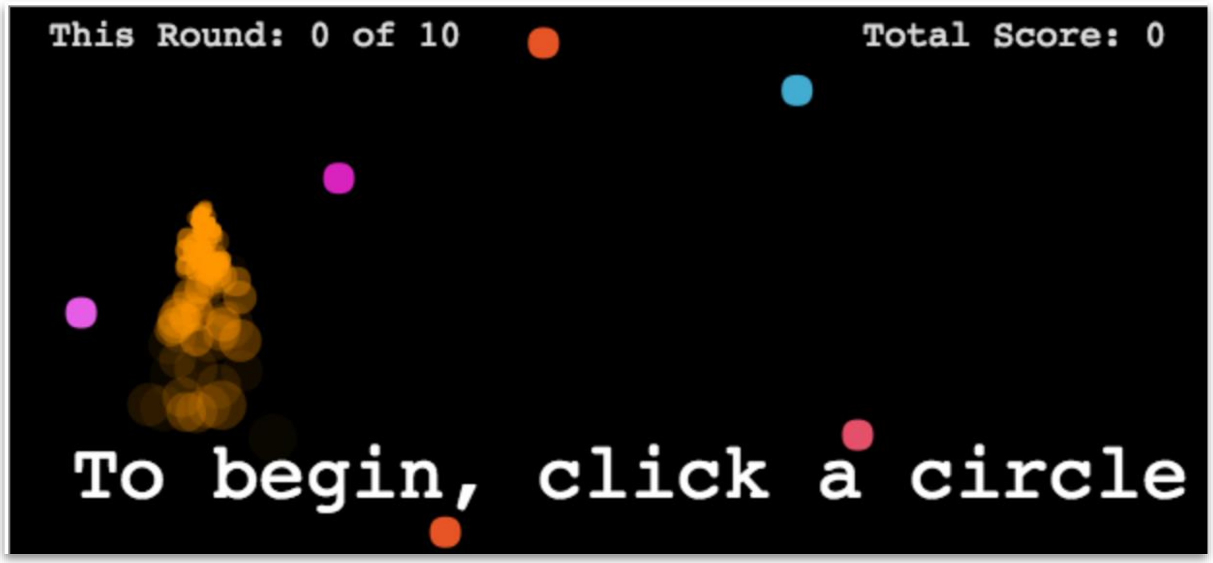
These are 2 common use cases of particle systems.

7) Now initialize the exhaust in `main.init()`:

```
this.exhaust = new this.Emitter();
this.exhaust.numParticles = 100;
this.exhaust.red = 255;
this.exhaust.green = 150;
this.exhaust.createParticles({x:100,y:100});
```

8) In `drawHUD()`, in the `GAME_STATE.BEGIN` block, add this code:

```
this.exhaust.updateAndDraw(this.ctx,{x:100,y:100});
```



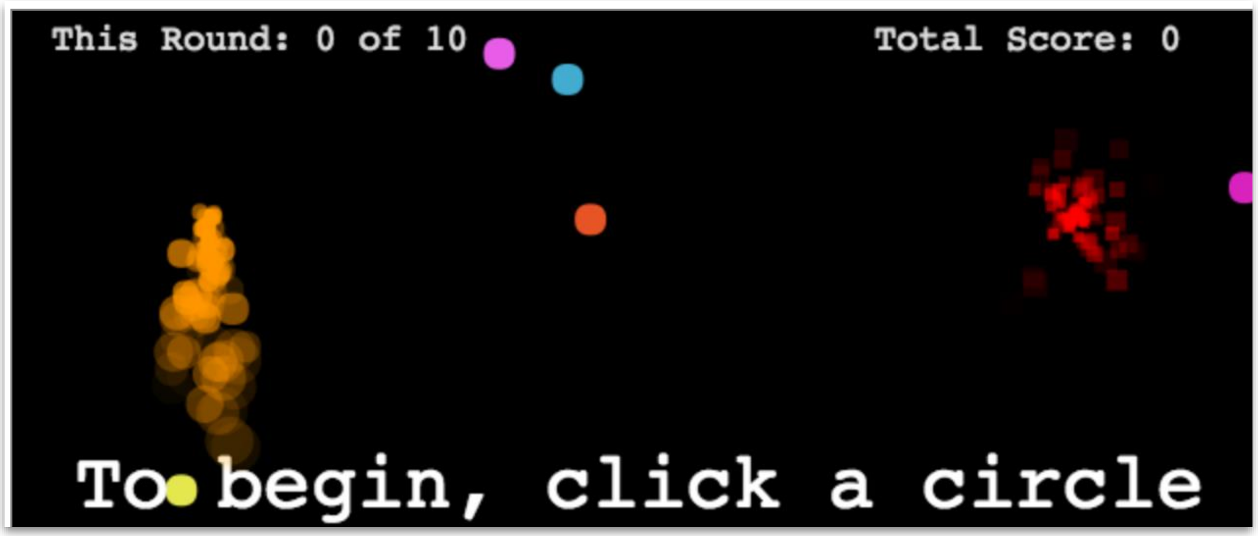
9) Reload the page and you should see your exhaust:

10) Here's the initialization code for the pulsar:

```
    // right side is "pulsar"
    this.pulsar = new this.Emitter();
    this.pulsar.red = 255;
    this.pulsar.minXspeed = this.pulsar.minYspeed = -0.25;
    this.pulsar.maxXspeed = this.pulsar.maxYspeed = 0.25;
    this.pulsar.lifetime = 500;
    this.pulsar.expansionRate = 0.05;
    this.pulsar.numParticles = 100;
    this.pulsar.xRange=1;
    this.pulsar.yRange=1;
    this.pulsar.useCircles = false;
    this.pulsar.useSquares = true;
    this.pulsar.createParticles({x:540,y:100});
```

11) Here's the drawing code:

```
this.pulsar.updateAndDraw(this.ctx,{x:540,y:100});
```



12) Reload the page:

13) Let’s have some fun and add a particle effect to all of the circles.
Warning: this will crash our frame rate pretty quickly once we start adding circles!

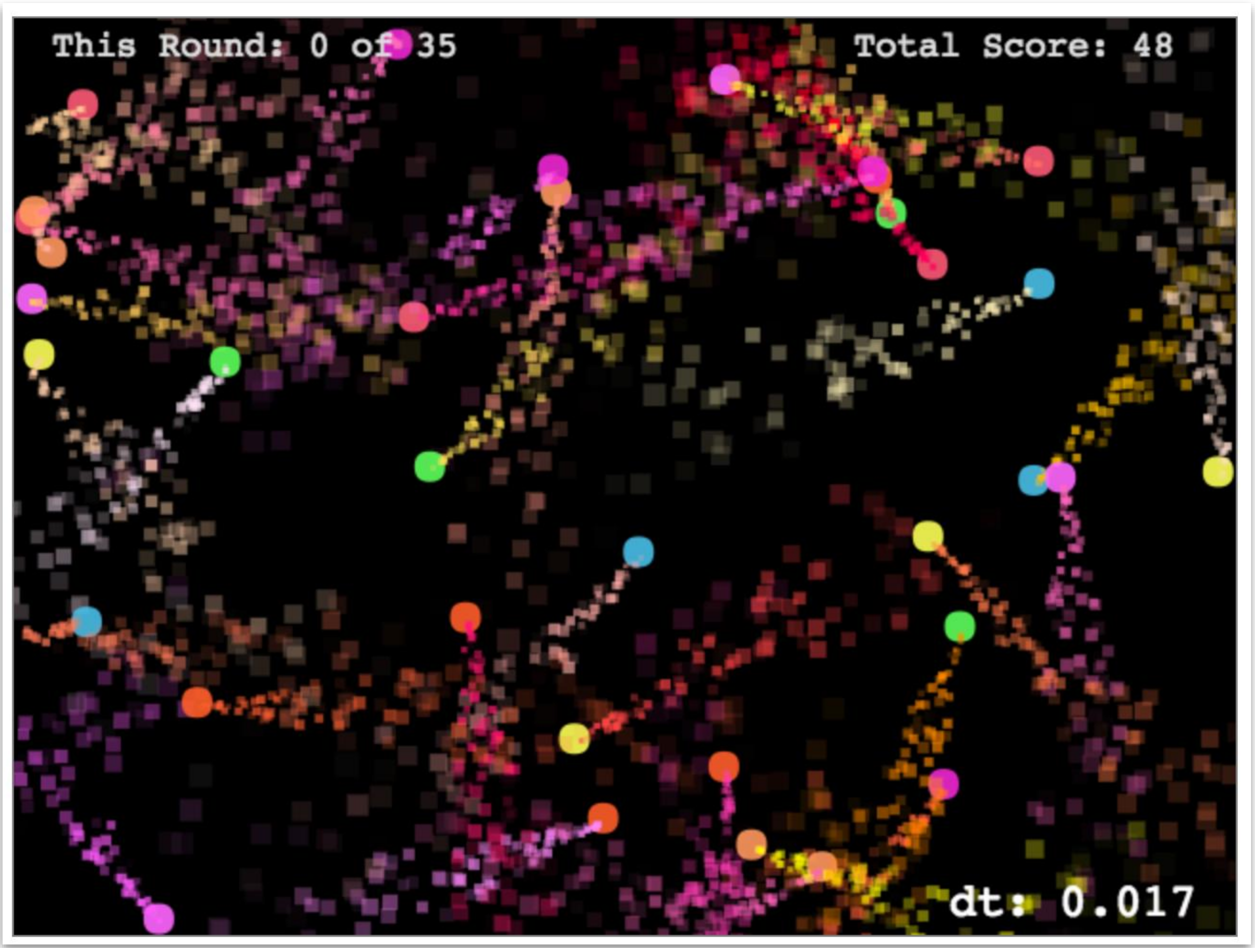
A) Add the following to the `for` loop in `makeCircles()`:

```
var pulsar = new this.Emitter();
pulsar.red = 255;
pulsar.green = Math.floor(getRandom(0,255));
pulsar.blue = Math.floor(getRandom(0,255));
pulsar.minXspeed = pulsar.minYspeed = -0.25;
pulsar.maxXspeed = pulsar.maxYspeed = 0.25;
pulsar.lifetime = 500;
pulsar.expansionRate = 0.05;
pulsar.numParticles = 100; // you could make this smaller!
pulsar.xRange=1;
pulsar.yRange=1;
pulsar.useCircles = false;
pulsar.useSquares = true;
pulsar.createParticles({x:540,y:100});
c.pulsar = pulsar;
```

Now every circle has a pulsar.

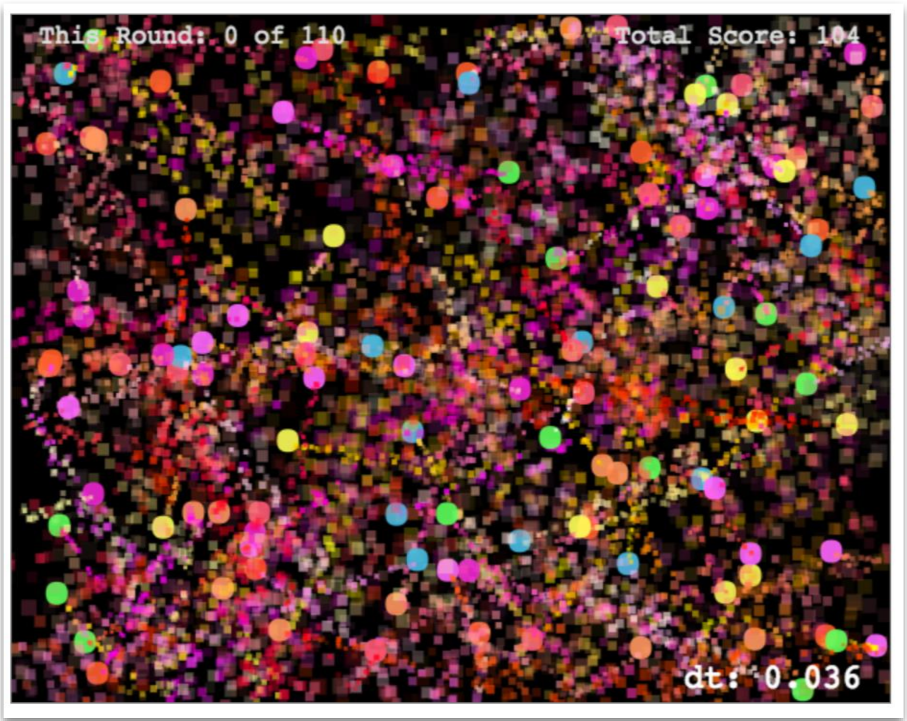
B) You'll need to draw the pulsars every time you draw the circles. In the `for` loop in `drawCircles()`, add the following:

```
if (c.pulsar){
      c.pulsar.updateAndDraw(ctx,{x:c.x,y:c.y})
}
```

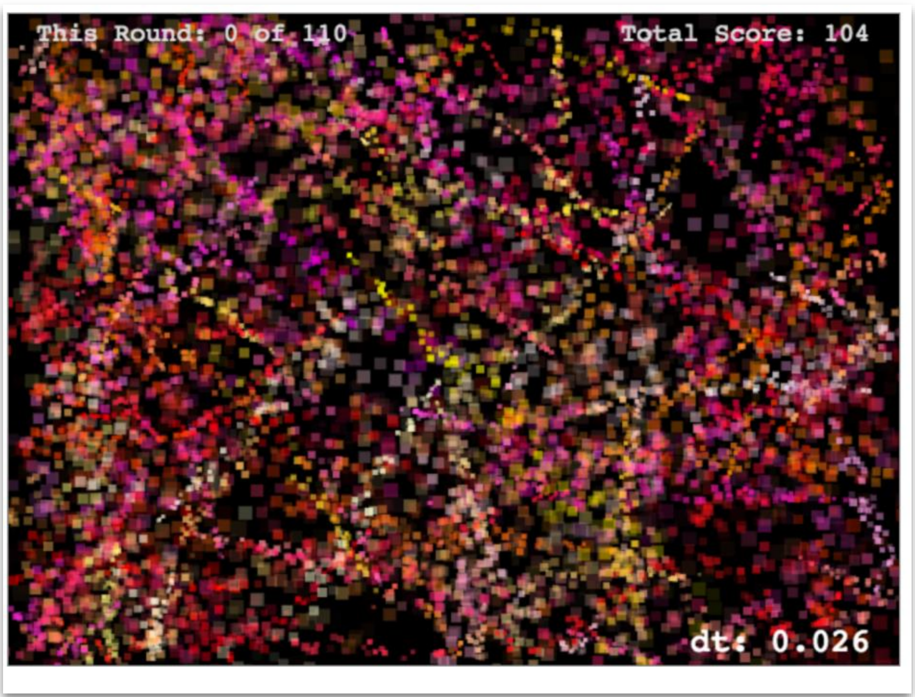


These particle trails have no game effects, but they sure look nice.

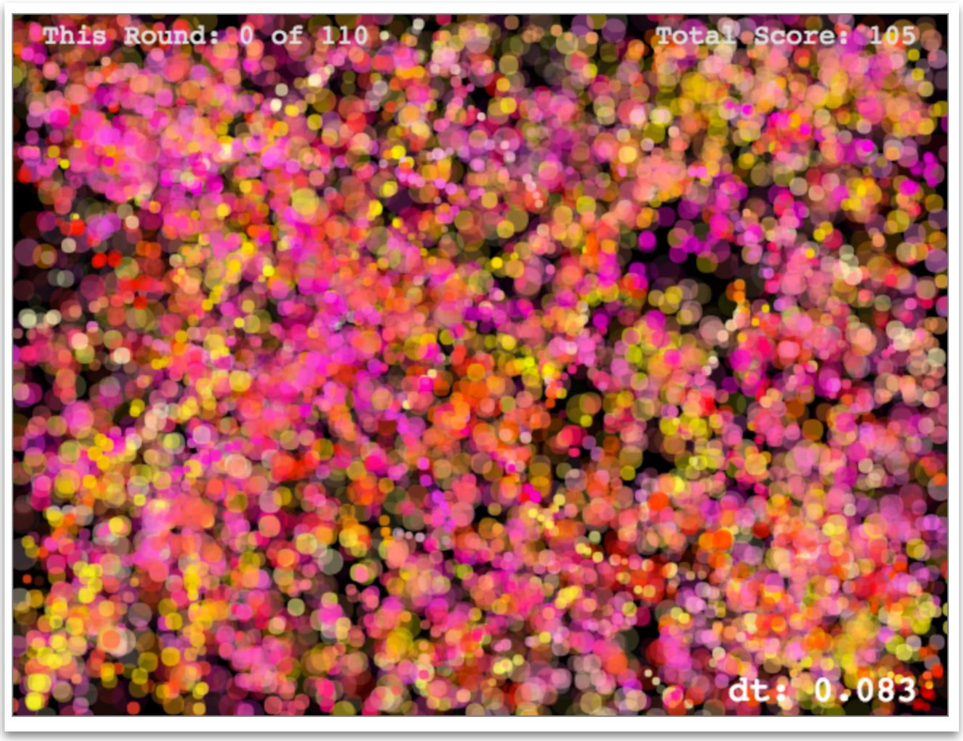
Note that 110 circles gives a really nice effect, but our frame rate has already dropped to 30 FPS. This is because each circle has an emitter, and each emitter has 100 objects - for a grand total of 11110 objects! trying to get moved and drawn 60 times a second.



Commenting out the `c.draw()` call gave a nice effect with no circles:



Here’s the same thing, just with circle particles instead of squares:



## Boomshine G Rubric

| DESCRIPTION | SCORE | VALUE % |
|---|---|---|
| **Basic Button** – Basic HTML Button created and shown correctly. | | **10** |
| **Canvas Button** – Custom button drawn and advances the level on click. | | **20** |
| **Object.freeze** – Object.freeze is used on an object to store your game constants. | | **15** |
| **Key Handler Module** – Key presses are now handled correctly in a module. | | **10** |
| **Key Presses in Module** – Key presses still work. Presses correctly handled in the key handler module. | | **15** |
| **Particle Pulsar** – Particle pulsar works correct to create exhaust effect. | | **15** |
| **Particle Trail** – Circles have a working particle trail. | | **15** |
| **Errors Thrown** – Any errors thrown in the console. | | **-20% (this time)** |
| **Additional Penalties** – These are point deductions for missing submission links, poorly written code, etc. There are no set values for penalties. The more penalties you make the more points you will lose. | | |
| | | |
| **TOTAL** | | **100%** |

**Zip up your code and submit it to the dropbox. Include a link to your work on Banjo in the submission comments.**