How to create beautiful effects with Mo.JS

Today, the article will look at mo.js, one of the new applications for creating beautiful animations from code. The article will cover a few basic functions, before creating a series of beautiful reaction effects.

If you want to start your own website, beautiful effects (animations or effects) can make it shine. There are many ways to achieve this, simply create an animated GIF from an existing video, to learn how to create an effect from scratch with software like Blender or Maya.

There are also libraries available for programmatic effects. Historically, web developers have used jQuery to create simple effects, but when the web was developed and HTML5 became the new standard, new options also appeared. CSS library for effects becomes extremely powerful under the new framework, along with JavaScript libraries specially designed for vector effects in the browser.

Today, the article will talk about mo.js, one of the new applications for creating beautiful animations from code. The article will cover a few basic functions, before creating a series of beautiful reaction effects.

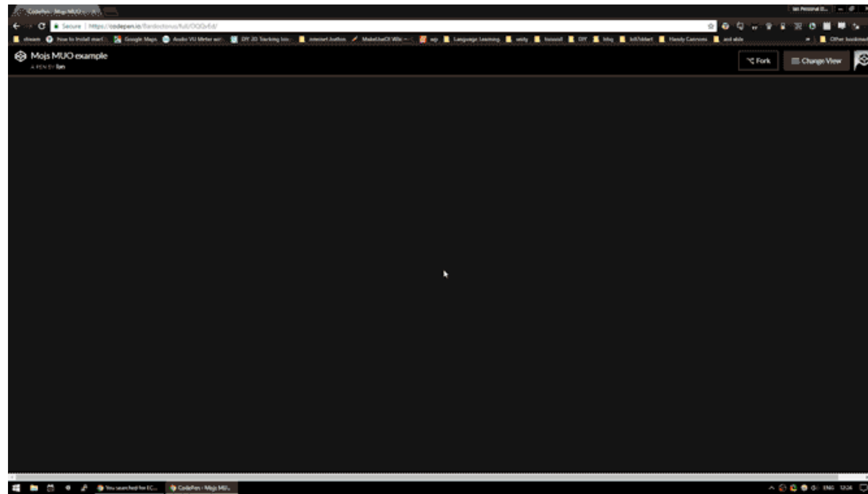
How to create beautiful effects with Mo.JS

1. About Mo.js
2. Create basic shapes with Mo.js
3. Basics of motion
4. Sort and reduce by Mo.js
5. Exploding with Mo.js
6. Major Events
7. Start creating psychedelic effects
8. Pentagon shapes 'know how to dance'
9. A little random
10. Amazing lines
11. These smart squares

About Mo.js

Mo.js is a library for creating motion graphics for the web easily. It is designed to create beautiful and simple templates for people who don't know anything about code, while still allowing veteran programmers to discover an artistic aspect they never knew. The article uses CodePen because it can do everything in the same window. You can use other code editors if you want. The full code is available at the following link:

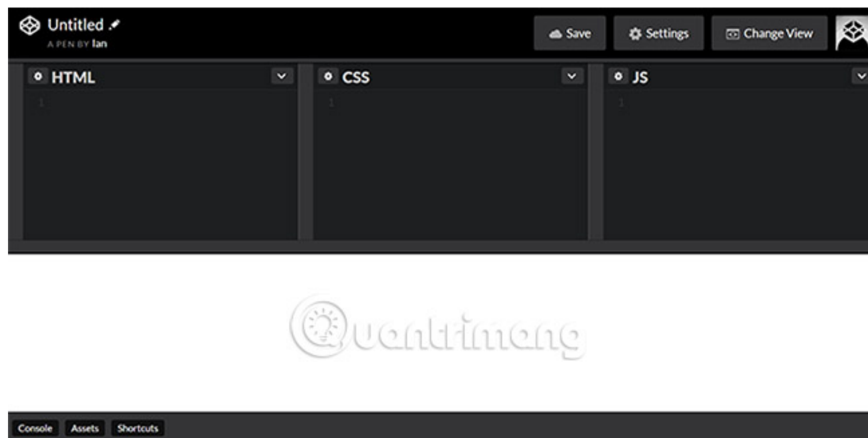
Before continuing, consider what the article will create today:



The article will use CodePen for today's project, as it allows working on everything in the same browser window. If you like, you can work in an alternative editor of your choice. If you want to skip step by step instructions, you can find the full code at:

```
https://static.makeuseof.com/wp-content/uploads/2018/02/MUO_mojs_tutorial_full_
```

Set up a new Pen and you will be greeted with this screen:



Before you start, you need to make some changes. Click the **Settings** icon at the top right and navigate to the **JavaScript** tab **.**

The article will use Babel as a code preprocessor, so choose Bebel from the menu. Babel makes JavaScript a little easier to understand, along with providing ECMAScript 6 support for older browsers.

You also need to import the mo.js library into the project. Do this by searching for mo.js in the **Add External Scripts** prompt **/ Pens text** and selecting it.

Add External Scripts/Pens

Any URL's added here will be added as <script>s in order, and run *before* the JavaScript in the editor. You can use the URL of any other Pen and it will include the JavaScript from that Pen.

Finally, click **Save** and **Close.** You are ready to start the next steps!

Create basic shapes with Mo.js

Before starting with the graphics, start with blurring the white background in the window. Change the background color attribute by writing this short code in the CSS window.

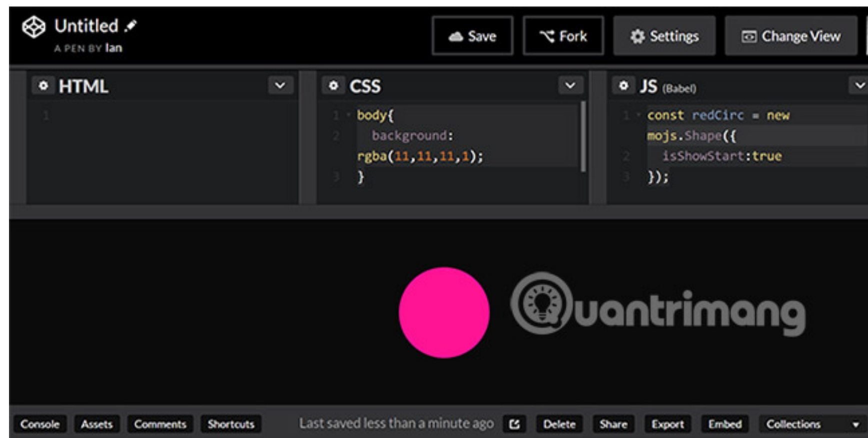
```
body {
background: rgba (11,11,11,1);
}
```

Shape creation is a simple process, and the concept behind it promotes the entire library. Please set the default circle. Enter this code in the JS window:

```
const redCirc = new mojs.Shape ({
isShowStart: true
});
```

Here, we will create a const value with the name redCirc and assign it to the new **mojs.Shape** . If you're completely ignorant of programming, pay attention to the parentheses here and don't forget the semicolon at the end!

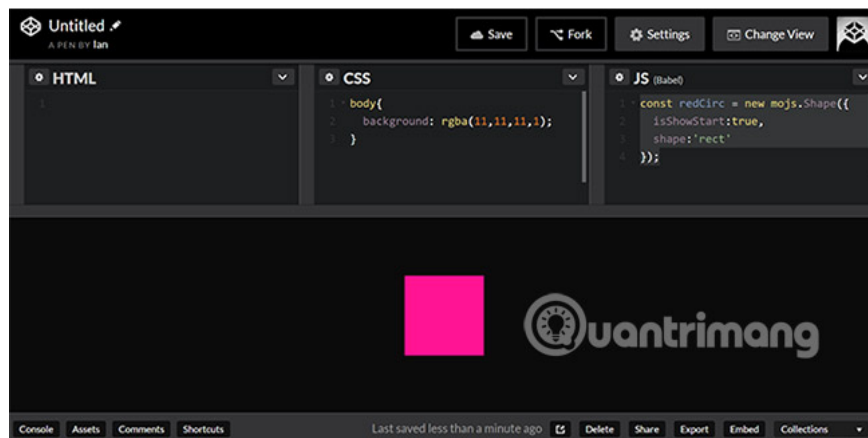
Until now, **isShowStart: true** is the only parameter, meaning that it will appear on the screen before you assign it any motion. You will see that it places a pink circle in the center of the screen:



This circle is the default **Shape** for mo.js. You can change this shape easily by adding the following line to the code:

```
const redCirc = new mojs.Shape ({
isShowStart: true,
shape: 'rect'
});
```

To add other attributes to an object, use commas to separate each attribute. Here, the article has added a shape attribute and identified it as **'rect'.** Save your Pen and you will see the default shape change to a square instead.



The process of passing values ??to Shape objects is how we customize them. Right now, we have a basic square. Try doing something.

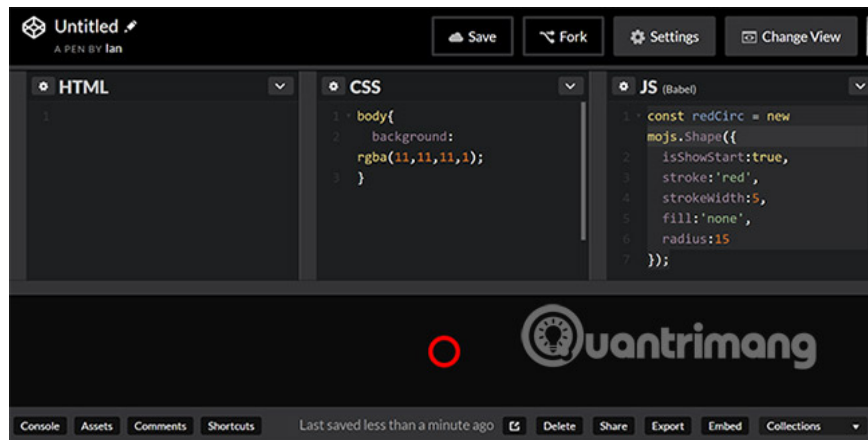
Basics of motion

To get something a little more impressive, set a red and blank circle on the inside.

```
const redCirc = new mojs.Shape ({
isShowStart: true,
stroke: 'red',
strokeWidth: 5,
fill: 'none',
radius: 15
```

```
});
```

As you can see, the author has assigned the width value to the circle's border and radius. Everything began to look a bit different. If the shape you created doesn't change, make sure you don't forget any commas or parentheses around the word **'red'** or **'none'** and make sure you've clicked **Save** at the top of the page.



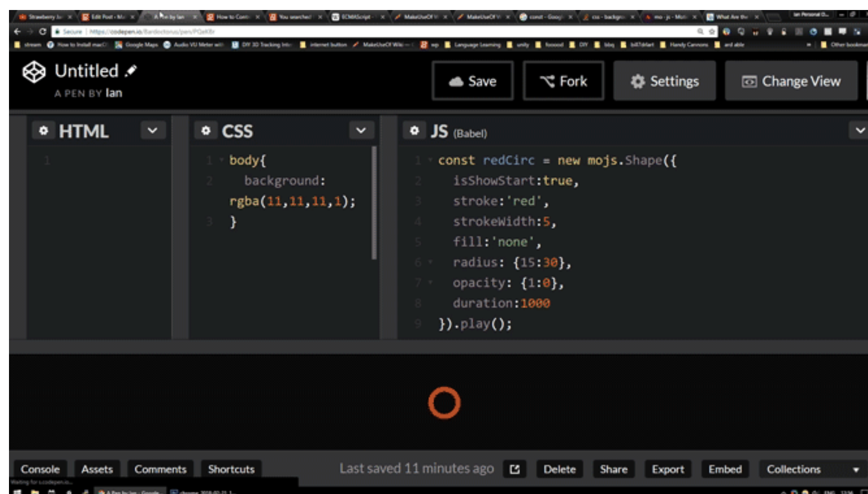
Let's add an effect to it. In the example above, this red circle appears where the user clicked, before fading. We can do this by changing the radius and opacity over time. Please modify the code as follows:

```
radius: {15:30},
opacity: {1: 0},
duration: 1000
```

By changing the radius attribute and adding the opacity and duration attributes, you gave instructions for the shape to perform over time. These are Delta objects, keeping the start and end information for these attributes.

You will notice that nothing happens. This is because you have not added the .play () function to request your instructions. Add it between brackets and semicolons, and you'll see your circle come alive.

Now, to make it really special, consider a few more specialized possibilities.



Sort and reduce by Mo.js

Right now, as soon as the circle appears, it starts to fade. This is perfectly fine, but we need to control it more.

You can do this with the .then () function. Instead of changing the radius or opacity, keep everything at the original position, before they change after a certain amount of time.

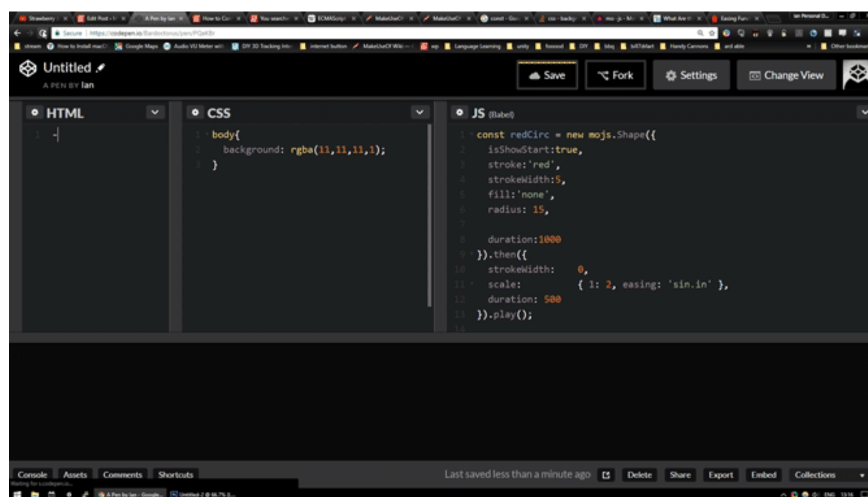
```
const redCirc = new mojs.Shape ({
isShowStart: true,
stroke: 'red',
strokeWidth: 5,
fill: 'none',
radius: 15,
duration: 1000
}) then ({
// due more stuff here
}). play ();
```

Now, your shape will appear with the assigned values, a delay of 1000 ms, before doing whatever you put in the .then () function. Please add some instructions between square brackets:

```
// due more stuff here
strokeWidth: 0,
scale: {1: 2, easing: 'sin.in'},
duration: 500
```

This code introduces another important part of the effect. Where the scale instruction changes from 1 to 2, we have also specified the sine wave based on reducing sin.in. Mo.js has various built-in curves, along with the ability for advanced users to add their own curves. In this case, time scale occurs along the sinusoidal curve.

For different curves with downward sinus lines, check **easings.net.** Combining this with the **strokeWidth** changes to 0 to the set time period and you will have a much more flexible disappearance effect.



Shapes are the basis for everything in Mo.js, but they are just the starting point of the whole process. Please continue with the burst (spark).

Exploding with Mo.js

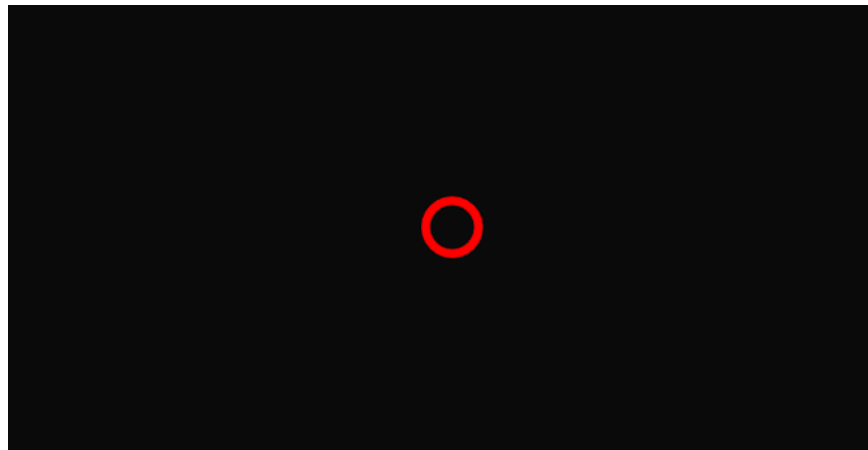
A spark in Mo.js is a set of shapes emanating from a central point. This will be the basis for the final effect. You can call a default spark the same way you create an original shape. Create some 'sparks' with:

```
const sparks = new mojs.Burst ({
}). play ();
```

You can see, just by adding a **burst** object empty and asking it to play, you will get the default spark effect. You can change the size and rotation speed of the spark by moving its radius and angle properties:

```
const sparks = new mojs.Burst ({
radius: {0:30, easing: 'cubic.out'},
angle: {0: 90, easing: 'quad.out'},
}). play ();
```

For example added a custom radius and rotation effect to the spark:

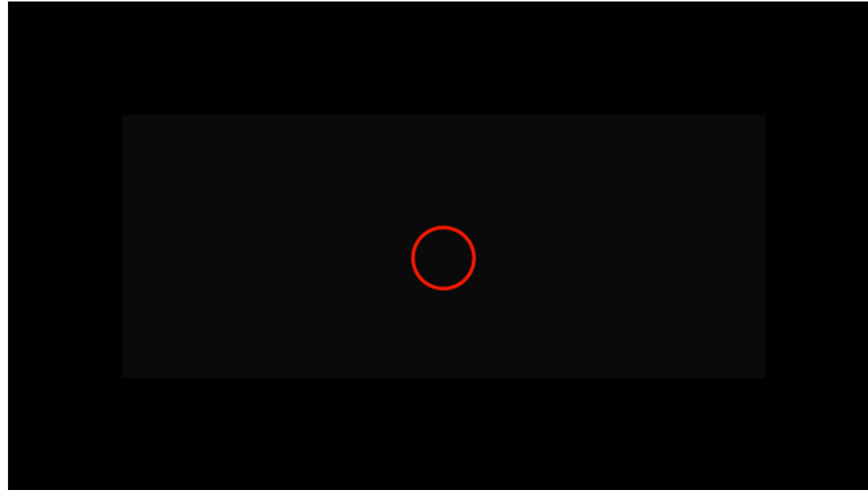


To make them look more like sparks, change the shape of the sparks and the number of shapes that sparks create. You do this by changing the properties of the original spark.

```
const sparks = new mojs.Burst ({
radius: {0:30, easing: 'cubic.out'},
angle: {0: 90, easing: 'quad.out'},
count: 50,
children: {
shape: 'cross',
stroke: 'white',
points: 12,
radius: 10,
fill: 'none',
angle: {0: 360},
duration: 300
}
}). play ();
```

You will notice that the child attributes are the same as the shape properties you worked on previously. This time, the author chose a cross shape. All 50 shapes now have the same properties. Everything looks good! This is the first thing users will see when they click.

Although we can see that the red border of the original redCirc shape has existed for too long. Try changing the duration of both animations so that they fit together. The result will look like this:



There are still a lot of things to do to create the final product.

Major Events

We will use the event handler to activate the effect in the location where the user clicked. At the end of the code block, add:

```
document.addEventListener ('click', function (e) {
});
```

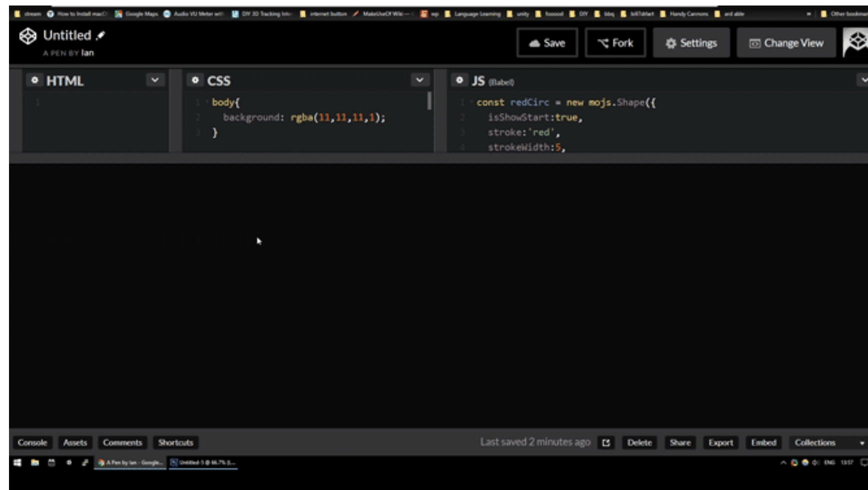
This code snippet detects clicks and executes any instructions within parentheses. You can add redCirc and variable sparks here.

```
document.addEventListener ('click', function (e) {
redCirc
.tune ({x: e.pageX, y: e.pageY,})
.replay ();

sparks
.tune ({x: e.pageX, y: e.pageY})
.replay ();
});
```

Your two functions are called .tune () and .replay (). The replay function is similar to the play function, but it specifies that the effect will start from the beginning whenever clicked.

The tune function passes values ??to the object, so you can change everything when it is activated. In this case, you are going through the page coordinates where the mouse is clicked, and assign the x and y position of the effect accordingly. Save the code and try clicking on the screen. You will notice a few problems.



First, the original effect is still displayed in the middle of the screen, even if the user does not click on anything. Second, the effect is not activated at the click point, but is on the right. We can fix both errors easily.

Shape and Burst have .play () functions at the end of their respective code blocks. You do not need this anymore because the .replay () function is being called in the event handler. You can delete .play () from both code blocks. For the same reason, you can also remove isShowStart: true, because you no longer need it to display from the beginning.

To fix the positioning problem, you will need to set the location values ??for the objects. If you remember when in the original shape, mo.js puts them in the middle of the page by default. When these values ??are combined with the mouse position, it will create a deviation. To eliminate this deviation, simply add these lines to both the redCirc object and the sparks variables:

```
left: 0,
top: 0,
```

Now the unique position value of the objects is the mouse position values ??transmitted by the event listener. Now things will work much better.

The process of adding this object to the event handler is how you will activate all effects, so remember to add all new objects to it from now on! Now that you have the basic knowledge of how you want effects to work, add some more explosive effects.

Start creating psychedelic effects

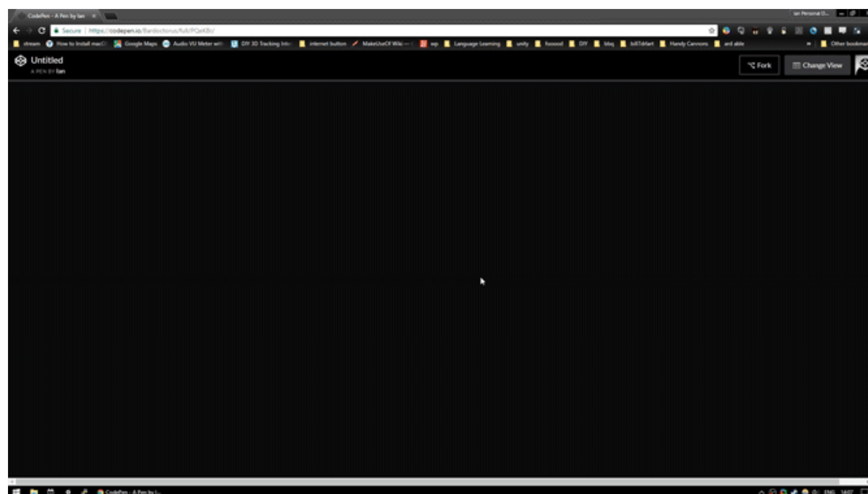
Let's start with some rotating triangles. The idea here is to create a stroboscopic hypnotic effect, and this setting is really very easy. Add another spark with the following parameters:

```
const triangles = new mojs.Burst ({
radius: {0: 1000, easing: 'cubic.out'},
angle: {1080: 0, easing: 'quad.out'},
left: 0, top: 0,
count: 20,
children: {
shape: 'polygon',
points: 3,
radius: {10: 100},
fill: ['red', 'yellow', 'blue', 'green'],
duration: 3000
}
});
```

Everything here is quite familiar, although there are a few new points to note. You will notice that instead of defining the shape as a triangle, for example call it a polygon before assigning the score it has 3.

The example also provided a fill function with a color array to operate, each fifth triangle would revert to red and the pattern would continue. High value for angle settings makes spins spin fast enough to create stroboscopic effects.

If your code doesn't work, make sure you've added the triangle object to the event listener class as you did with the previous object.



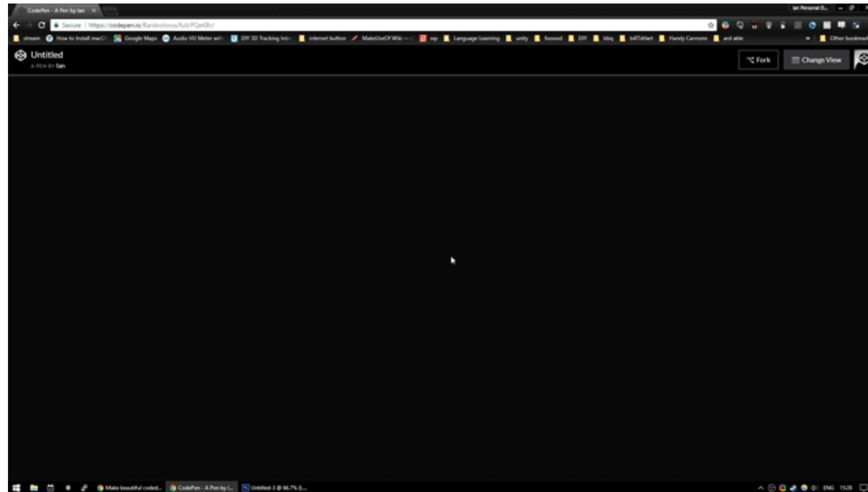
Great! Let's create some more bust.

Pentagon shapes 'know how to dance'

You can use something similar to a triangular object to create a spark following it. This slightly corrected code creates hexagons that rotate on top of each other in vibrant colors:

```
const pentagons = new mojs.Burst ({
radius: {0: 1000, easing: 'cubic.out'},
angle: {0: 720, easing: 'quad.out'},
left: 0, top: 0,
count: 20,
children: {
shape: 'polygon',
radius: {1: 300},
points: 5,
fill: ['purple', 'pink', 'yellow', 'green'],
delay: 500,
duration: 3000
}
});
```

The main change here is that you have added a 500ms delay, so the sparks will start after the triangular sparks. By changing a few values, you can make pentagonal shapes rotate in the opposite direction to the triangles. Incidentally, by the time the pentagon appears, the stroboscopic effect of the triangle makes us feel like they're spinning together.



A little random

Add effects using random values. Create a spark with these attributes:

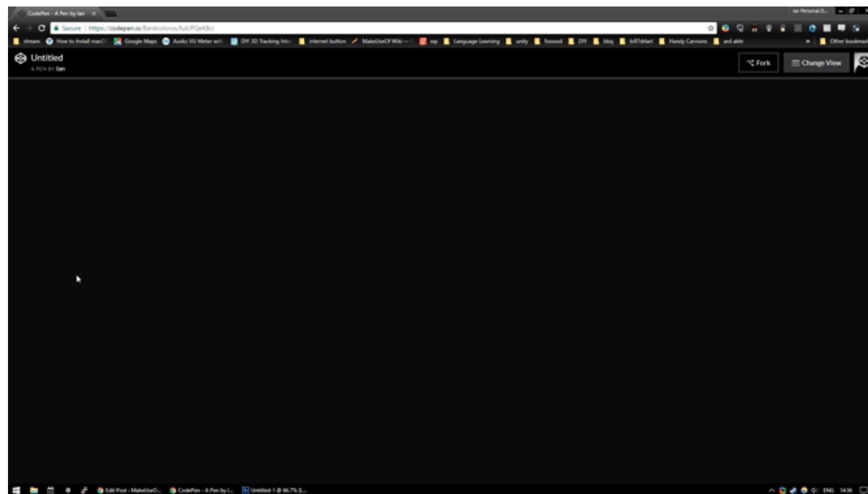
```
const redSparks = new mojs.Burst ({
left: 0, top: 0,
count: 8,
radius: {150: 350},
angle: {0:90, easing: 'cubic.out'},
children: {
```

```
shape: 'line',
stroke: {'red': 'transparent'},
strokeWidth: 5,
scaleX: {0.5: 0},
degreeShift: 'rand (-90, 90)',
radius: 'rand (20, 300)',
duration: 500,
delay: 'rand (0, 150)',
}
});
```

This explosion will create lines that start red and fade until they become transparent, smaller and smaller over time. What makes this component interesting is that random values ??are used to identify some attributes for it.

DegreeShift provides sub-objects with a starting angle. By randomly creating this, it creates completely different sparks every click. Random values ??are also used for radius and latency to add a chaotic effect (mixture).

This is the effect of itself:



Since you are using random values ??here, you need to add an additional method to the object's event handler:

```
redSparks
.tune ({x: e.pageX, y: e.pageY})
.replay ()
.generate ();
```

The generate () function calculates new random values ??every time an event is called. Without this, the shape will select random values ??the first time it is called and continue to use those values ??for every next call. This will completely ruin the effect, so make sure you add this function!

You can use random values ??for almost every element of mo.js objects and they are a simple way to create unique effects.

However, randomness is not the only way to add dynamic motions to the effect. Consider stagger function.

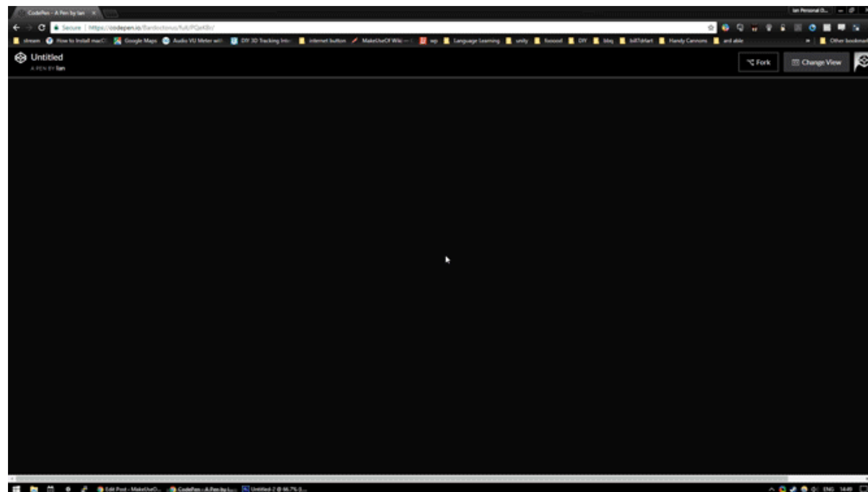
Amazing lines

To show how the stagger function works, the example will create something like a Catherine wheel. Create a new spark with the following parameters:

```
const lines = new mojs.Burst ({
radius: {0: 1000, easing: 'cubic.out'},
angle: {0: 1440, easing: 'cubic.out'},
left: 0, top: 0,
count: 50,
children: {
shape: 'line',
radius: {1: 100, easing: 'elastic.out'},
fill: 'none',
stroke: ['red', 'orange'],
delay: 'stagger (10)',
duration: 1000
}
});
```

Everything here is now familiar, a spark with 50 objects with red or orange lines. The difference is that you pass the latency attribute of the stagger function (10). This adds 10ms of latency between each object, creating the desired rotation effect.

The stagger function doesn't use any random values, so you won't need the constructor in the event handler this time. Take a look at all you have so far:



You can easily end here, but add something to make this project more complete.

These smart squares

For this last spark, create something with a rectangle. Add this object to your code and the event listener:

```
const redSquares = new mojs.Burst ({
radius: {0: 1000, easing: 'cubic.out'},
```

```
angle: {360: 0, easing: 'quad.out'},
left: 0, top: 0,
count: 20,
children: {
shape: 'rect',
radiusX: {1: 1000},
radiusY: 50,
points: 5,
fill: 'none',
stroke: {'red': 'orange'},
strokeWidth: {5:15},
delay: 1000,
duration: 3000
}
});
```

This object is similar to what you have done today. It helps you visualize how easy it is to create geometric patterns through code.

Once the code runs, you will see how amazing it is!

With the last object added, everything is complete. Let's see how all the objects work!

This simple introduction to mo.js includes the basic tools needed to create spectacular effects. The way tools are used can create almost anything, and for many tasks, web libraries are a simple option to use Photoshop, After Effects, or other software.

This library is useful for those working in both web development and programming, the event handler used in the project can easily be used to create buttons and feedback text on website or in application.

Good luck!

See more:

1. Instructions for creating animated GIF images in Adobe Photoshop
2. 5 effective online Gif image creation tools
3. How to edit GIF images using EZGIF tool online

You finished reading the article "**How to create beautiful effects with Mo.JS**" edited by the **TipsMake** team. We hope this article has provided you with many useful tech tips and tricks. You can search for similar articles on tips and guides. Thank you for reading and for following us regularly.

© 2019 TipsMake.com