

Learn HTML **Canvas** : Pixels & Physics

Essa é uma tradução livre do conteúdo produzido pelo canal **Franks Laboratory**, antes de entrarmos no material em si, gostaria primeiramente de agradecer em nome de todos que terão acesso a esse conteúdo, e que poderão estudar graças aos conhecimentos passados.

ATENÇÃO: utilize esse material apenas como fonte de estudos. Para utilização comercial contate diretamente o criador nas redes sociais abaixo transcritas. Qualquer tentativa de utilização deste material para fins lucrativos sem a autorização de quem efetivamente tenha propriedade sobre o conteúdo, resultará na aplicação das sanções e penas previstas em lei.

A imagem disponível para download que consta como exemplo neste documento, pode ser utilizada nos mesmos moldes do parágrafo anterior.

Não tenho propriedade intelectual sobre nenhum material contido neste documento, apenas o traduzo, todos os direitos estão reservados à seus respectivos criadores.

Desde logo peço desculpas por possíveis erros de português.

Se você gostou do conteúdo saiba que ele tem um curso sobre o tema, onde ele o trata com mais profundidade: <https://www.udemy.com/course/learn-html-canvas-pixels-particles-physics/?referralCode=F7977062A4BC964A1F5E>

Apoie também em suas redes sociais:

- Twitter: https://twitter.com/code_laboratory
- CodePen: <https://codepen.io/franksLaboratory>
- TikTok: https://www.tiktok.com/@franks_laboratory

link do vídeo: <https://www.youtube.com/watch?v=vAJEHf92tV0>

A imagem aqui utilizada pode ser encontrada no repositório deste projeto.

Siga e apoie a artista responsável em:

- instagram: https://www.instagram.com/leticia_loureiro_/

Também sigam este humilde tradutor em fase de desenvolvimento como programador:

- github: <https://github.com/arturneppel>

ÍNDICE

INTRODUÇÃO

2 SETUP BÁSICO DO PROJETO

- 2.1 HTML
- 2.2 CSS

3 CONVERTENDO IMAGENS EM CÓDIGO

4 COMO USAR O CANVAS

5 JAVASCRIPT CLASSES E O SISTEMA DE PARTÍCULAS

6 DESENHANDO RETÂNGULOS

7 DESENHANDO IMAGENS

INTRODUÇÃO

O Java Script é uma ferramenta extremamente poderosa, e vamos explorar os segredos da arte e animação no front-end e desenvolvimento web atual.

Nessa aula você vai começar aprendendo a desenhar um retângulo e uma imagem, e vai descobrir como transformar esses retângulos em um sistema de partículas inteligente que pode recriar e lembrar de formas, baseado no *pixel data* de qualquer imagem.

Será demonstrado como aplicar a cada partícula propriedades físicas como por exemplo atrito e maleabilidade, e como fazer essas partículas interagirem com o mouse, quebrando e remodelando sua forma, e ao final automaticamente voltarem às suas posições iniciais.

Em resumo você vai aprender a transformar imagens em arte interativa digital.

Apesar de ser um curso introdutório ao `<canvas>` é necessário ter conhecimentos básicos de Java Script bem como de programação.

Mas ainda sim o curso é bem explicativo então mesmo que você não esteja familiarizado com a sintaxe da linguagem, ainda é possível entender completamente.

1 SETUP BÁSICO DO PROJETO

1.1 HTML

Vamos começar criando os três arquivos bases para esse projeto. O `index.html` , `style.css` , e o `script.js` .

Se ainda não baixou a imagem que utilizaremos do projeto, faça o download aqui: [link](#)

Vamos começar com o arquivo `index.html` pois ele é a árvore dos quais os outros dois arquivos serão os galhos.

Primeiro faremos a estrutura básica do HTML, já *linkando* no arquivo os outros dois que criamos:

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>pixel</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>

  <script src="script.js"></script>
</body>
</html>
```

lembre-se sempre de deixar o arquivo do script por último dentro do `<body>` , pois vamos trabalhar os elementos dentro do script, logo eles devem ser carregados antes dele.

Vamos criar o elemento `canvas` através da tag `<canvas>` e atribuindo um identificador para ele:

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>pixel</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <canvas id="canvas1"></canvas> |<<<<<<<<<<-----

  <script src="script.js"></script>
</body>
</html>
```

O **canvas** é um elemento da HTML utilizado para desenhar em uma página web. Podemos desenhar tanto formas estáticas como formas dinâmicas, e ainda fazer animações interativas que reagem a *inputs* do usuário.

Então, em resumo, ele é um container para as nossas artes. É a nossa tela de pintura (literalmente **canvas** em inglês).

Podemos utilizar o Java Script para desenhar formas geométricas, linhas, textos e imagens na nossa tela. Parece muito simples, mas uma vez que você entender como o **canvas** funciona, vai conseguir fazer uma quantidade infinita de coisas com ele, como por exemplo: arte digital e jogos.

Vamos criar uma `<div>` com uma classe, e dentro um botão com um `id`.

```
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>pixel</title>
  <link rel="stylesheet" href="style.css">
</head>
<body>
  <canvas id="canvas1"></canvas>

  <div class="controls">
    <button id="warpButton">Warp</button>
  </div>
```

```
<script src="script.js"></script>
</body>
</html>
```

A ideia é que, quando clicarmos nesse botão, a imagem vai se desfazer em pequenas partículas e vai se remontar automaticamente.

Poderemos controlar a velocidade e o peso das partículas enquanto elas se movem, não se preocupe não é nada complicado.

1.2 CSS

No CSS vamos começar *resetando* a margem e o *padding* (*espaçamento interno*) de todos os elementos utilizando o seletor global `*`.

Vamos alterar a cor de fundo do nosso `canvas` para azul, apenas para enxergá-lo enquanto estivermos trabalhando com ele.

Na classe `.controls` vamos trocar seu `position` para `absolute`. Isso vai retirar esse elemento do *flow* da página e vai posicioná-lo sobre o `canvas`.

Precisamos ter certeza de que nosso botão não vai ser coberto por nenhum outro elemento e sempre será clicável, então vamos atribuir um `z-index` com valor de `100` para ele.

Também vamos adicionar o mesmo `display` para o `canvas`.

```
*{
  margin: 0;
  padding: 0;
}

#canvas1 {
  background-color: blue;
  position: absolute;
}

.controls {
  position: absolute;
  z-index: 100;
}
```

E a nossa página está assim:



2 CONVERTENDO IMAGENS EM CÓDIGO

Para conseguirmos adicionar uma imagem no nosso `canvas`, primeiro precisamos carregá-la na nossa página. Para isso vamos utilizar o elemento `` disponível na HTML.

[illegible]

Mas como você percebeu o `src` da imagem permanece vazio. Vamos precisar fazer um pequeno truque, e vamos adicionar nossa imagem de uma forma um pouco diferente.

Primeiro localize a imagem que foi baixada no início deste documento. É uma imagem normal em formato `png`. Se simplesmente informássemos a localização dessa imagem em no atributo `src`, como fazemos normalmente, nós conseguiríamos renderizar essa imagem no nosso `canvas`, mas não conseguiríamos analisar para o `pixel-data` e quebrar ela em várias partículas.

Além disso, Se tentássemos fazer isso da forma convencional (simplesmente adicionando a localização da imagem), receberíamos um erro: `cavas was tainted by cross-origin data`. esse erro só vai acontecer enquanto estivermos rodando nosso código de forma local. Quando você der `upload` no seu projeto em um servidor online, o arquivo `png` no `scr` será considerado com a mesma origem. Mas nós queremos que esse projeto funcione desde logo, ou seja, não queremos ter nenhum erro mesmo quando trabalhamos localmente como agora.

Para “burlar” esse problema, nós vamos converter a imagem em código.

Com o código em mãos, vamos colocá-lo no `src` do elemento ``, e dessa forma a imagem será considerada parte integrante do HTML e não um arquivo localizado em outro lugar. Assim, terá a mesma origem em qualquer circunstância.

Muitas pessoas não sabem que imagens que estão na internet podem ser quebradas em uma única, e muito longa, linha de código.

Chamamos essa linha extensa de código de `BASE64 STRING` que nada mais é do que uma linha de código que contem toda a informação da imagem e substitui o arquivo dela por completo.

Para fazer isso, ou seja, converter nossa imagem para `BASE64 FORMAT`, desenhando ela no `canvas`, e depois chamando dois métodos internos dele, tornando o `canvas` inteiro, com a imagem desenhada nele, em uma `string` de dados.

Hoje vamos utilizar uma opção mais fácil para fazer esse procedimento, e utilizar um dos diversos websites disponíveis para fazer esse trabalho para nós. Basta procurar `png to base64`. e provavelmente o primeiro site já vai fazer isso por você.

O nosso código fica assim:

```
data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAwAAAPuCAyAAABHL4GLAAACXBIWXMAAASAAALEgHS
3X78AAAgAELEQVR4nOydB3wb9dnHn9Pesmx5D3lm2U6cqUAgo0CAFkoAA28pI1CgbAIts31fRtkdhJa2oWUEKC3Dpa
FQVhkJ2xBIyGTEiZXh2Lg8h2QN3/v5n+50J1nb2n6+re070+nur50Q/797nt/zUDRNA4IgCIJEgtnUnAMATexPjuAp
W8lPq6WlHS8kgiBI9oBiAUEQBAMJ2dS8CgCWAwd5bQqzuwUA1gPA2LZLSz9eWQRBkMwGxQKCIAGyAb0pmUQ01rACQ
R/DFRpgBcMdeHURBEEyFqQLCIIgCIPZ1FwJAKvZn4ARBINR2181o0RcYjIq5SqxhAIAMVQBPYeHD2z+aJey50i/0e8
pX5HjtVpatuJVRhAEyTxQLCAIgxKxZKmTiAsC3QLkmoLHGw1ebLy2nyQKST89tw8A0yrq4PcvFygKIrZ1t3ZB0898
nLX5g93Fgo0McAKhg1T/VojCIJkGigWEARBPiCCKMkaQGLGhWU5UF1fDJxAoLgHKIDc3Fyoq6uBPGMevz8nFrjLXVv
b4De/XD8wMmQTHvuiVkvLevy8IQiCZA4oFhAEQaYQZlPzclYkX0j/qtU6BVTPKoaahLQ60XeByjm/6BUKqFuWg2U1
pV6H/ITCcLlkWEb3Hzx73q7u/pyBadBwYagCJJBoFhAEASZArAVjdYESjUqq8lnREJ5TT4vDIDy/m2QSCVMJKGyymN
jCCUQAi3/7v+e7m3dtE0oGFa0Wlo24uc0QRAK/UGxgCAIkswwfoQ7/A3LUpmErp5VTM2cwwFqvcL7gJ9YMFVWQG1dD
chkUu8uAUQBSTt6veVDWHsAyw4ph7UGmUowUA8DMvR9IwgCJL+oFhAEATJQoKJBLVWMD64uFpEogky0etFoAQ7sGJ
Bq9NA4+xZoNPrvA8FEAnE0LzugRdg91d7+ccef/WuCWKBLF919t293Z18StJXrGDAXgwIgiBpjATfHARbkOwhhEhwn
S6ullTPLBExG6jAN4ok...
```

Esse é só o início pois a linha é gigantesca. Para saber se a conversão foi feita corretamente basta ver se a linha começa com `data:image/png` e provavelmente ocorreu tudo bem;

colocando dentro do atributo `src` e recarregando a página temos:



Agora que temos a imagem renderizada na página podemos trabalhar com ela no JavaScript.

Então o que fizemos até agora foi: ao invés de apontar simplesmente onde a imagem está no computador, nós a convertemos no formato `BASE64` e adicionamos diretamente no atributo `src` da tag ``, e agora ela será entendida como parte do código do HTML ao invés de um arquivo separado. Assim vamos conseguir trabalhar da forma que quisermos com ela no Java Script.

Como queremos desenhar essa imagem no nosso `canvas` e não na página, nós vamos dar `display: none` pra ela no CSS a escondendo.

```
img {  
  display: none;  
}
```

Pronto, agora a imagem está escondida, mas podemos usar o Java Script para desenhá-la quando quisermos.

3 COMO USAR O CANVAS

Vamos abrir primeiramente nosso arquivo de script, e vamos criar um evento de `load`.

O evento de `load` vai garantir que o nosso script só vai começar a ser executado quando todos os elementos e arquivos (como por exemplo, arquivos CSS, outras imagens, vídeos etc), forem carregados.

OBS: também podemos adicionar essa configuração com o atributo `defer` dentro da tag `script` porém vamos fazer dessa forma seguindo o tutorial.

Temos que adicionar esse detalhe, pois o código JavaScript é executado mais rápido do que a nossa imagem é carregada, então podemos acabar com um `canvas` em branco.

Selecione o nosso `canvas` com o método `getElementById` e passar o id do nosso `canvas` para ele.

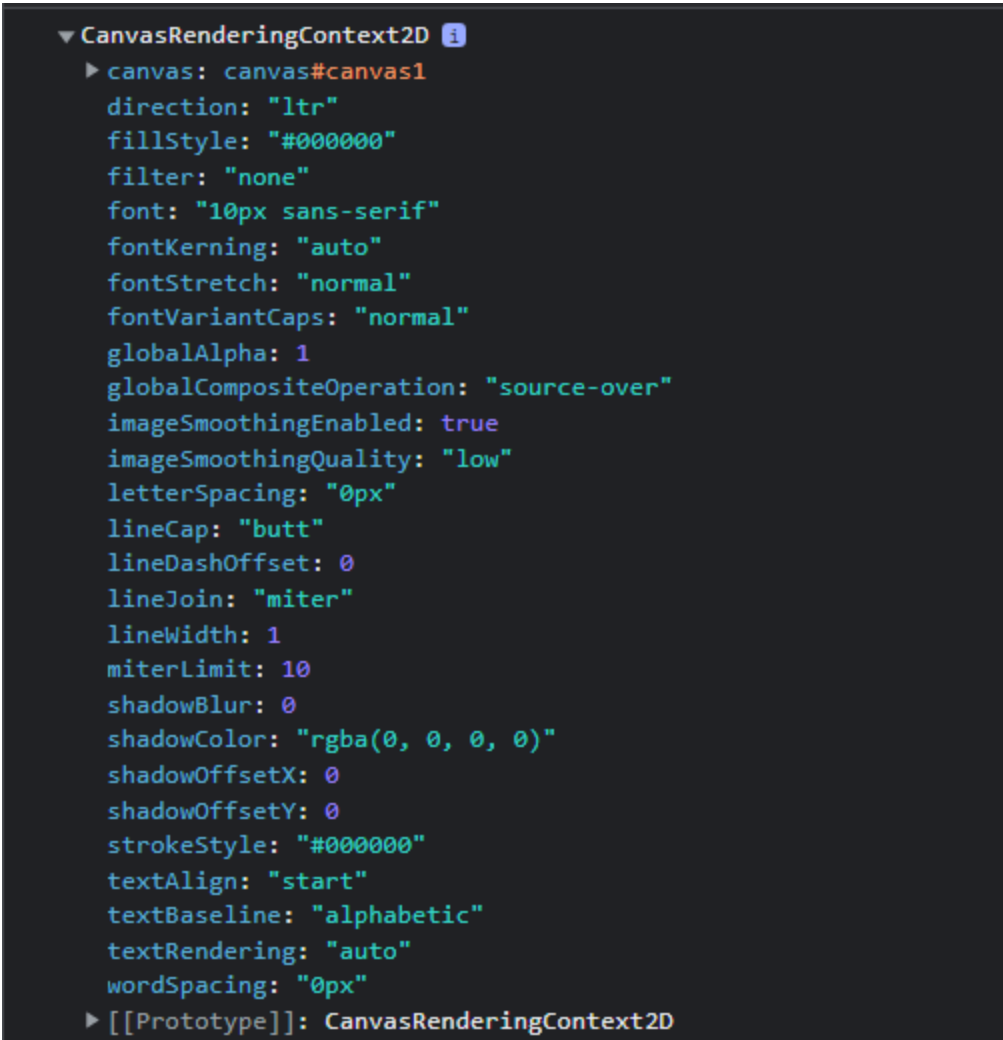
Agora vamos dar um contexto ao nosso `canvas`, através do método `getContext('2d')`. Esse é um método especial que só pode ser usado em uma variável que guarda a referência de um elemento `canvas` do HTML5.

O `getContext` recebe um argumento, que chamamos de `context type` que pode receber `2d` ou `webgl`. Vamos hoje usar o `2d`.

Quando for chamado dessa forma, o `getContext` criará uma nova instância de um objeto chamado `canvas rendering context 2d`. Esse objeto por sua vez, tem todos os métodos e configurações necessárias que nós precisaremos para fazer nosso projeto de hoje.

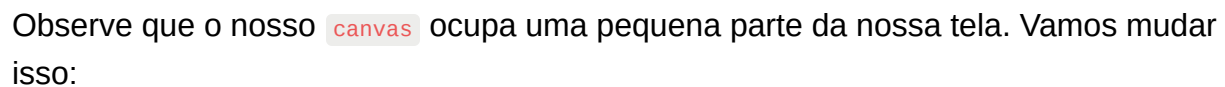
```
window.addEventListener('load', function(){
  const canvas = document.getElementById('canvas1');
  const ctx = canvas.getContext('2d');
})
```

Se pedirmos para o console exibir o que é o `ctx`, (através do `console.log(ctx)`) vamos ver o seguinte



```
▼ CanvasRenderingContext2D ⓘ
  ▶ canvas: canvas#canvas1
  direction: "ltr"
  fillStyle: "#000000"
  filter: "none"
  font: "10px sans-serif"
  fontKerning: "auto"
  fontStretch: "normal"
  fontVariantCaps: "normal"
  globalAlpha: 1
  globalCompositeOperation: "source-over"
  imageSmoothingEnabled: true
  imageSmoothingQuality: "low"
  letterSpacing: "0px"
  lineCap: "butt"
  lineDashOffset: 0
  lineJoin: "miter"
  lineWidth: 1
  miterLimit: 10
  shadowBlur: 0
  shadowColor: "rgba(0, 0, 0, 0)"
  shadowOffsetX: 0
  shadowOffsetY: 0
  strokeStyle: "#000000"
  textAlign: "start"
  textBaseline: "alphabetic"
  textRendering: "auto"
  wordSpacing: "0px"
  ▶ [[Prototype]]: CanvasRenderingContext2D
```

Hoje vamos utilizar alguns básicos e nos aprofundar muito no método `drawImage`.



com essas duas propriedades estamos definindo que o nosso `canvas` terá o tamanho todo disponível.

Cada partícula é um pequeno elemento gráfico, que pode ser uma imagem ou um formato. nós podemos programar essas partículas para que se pareçam e se

comportem de determinada maneira simulando os mais diversos tipos de efeitos.

Por exemplo, poderíamos querer um efeito de fogo, bolas quicando, hordas de inimigos em um jogo e muitas outras coisas.

Hoje a ideia é fazer cada partícula ser um pixel na nossa imagem, e nós vamos quebrar nossa imagem em peças individuais. Ao mesmo tempo, nós também vamos fazer com que essas partículas reajam ao mouse com propriedades físicas como atrito e maleabilidade.

para fazer isso vamos precisar criar uma classe no JavaScript chamada `Particle` com o `p` maiúsculo.

Classes são como se fossem as plantas de uma casa, ou seja, um modelo, para que possamos criar diversos objetos que são seus filhos.

O JavaScript é uma linguagem baseada em protótipos. A **programação baseada em protótipos**

é um estilo de programação orientada a objetos em que a reutilização de comportamento (conhecida como herança) é realizada por meio de um processo de reutilização de objetos

existentes que servem como protótipos. Esse modelo também pode ser conhecido como programação *prototípica, orientada a protótipos, sem classes ou baseada em instância*.

Não se preocupe se ficou confuso com essa parte, mas procure entender mais sobre.

Como o JavaScript é uma linguagem baseada em protótipos, todo objeto em JavaScript possui uma propriedade interna chamada `prototype` que pode ser utilizada para estender as propriedades do objeto e seus métodos.

Classes em JavaScript podem ser consideradas um `syntactical sugar`, que é uma forma mais elegante e limpa de escrever o código baseado no sistema nativo de herança proporcionado pelo `prototype`, que imita outras classes de outras linguagens de programação.

Simplificando, uma classe é um modelo ou em inglês **template**.

Criando a classe:

```
window.addEventListener('load', function(){
  const canvas = document.getElementById('canvas1');
  const ctx = canvas.getContext('2d');
```

```

    canvas.width = window.innerWidth;
    canvas.height = window.innerHeight;

    class Particle { |<<<<<-----
    }
  })

```

Toda vez que chamarmos essa classe que criamos ela vai criar um novo objeto `particle` para nós.

Vamos criar uma outra classe também chamada `Effect`. Essa classe vai cuidar do efeito inteiro, mexendo com todas as partículas de uma vez.

A última coisa que vamos precisar aqui é um loop de animação que vamos fazer através de uma função.

```

window.addEventListener('load', function(){
  const canvas = document.getElementById('canvas1');
  const ctx = canvas.getContext('2d');
  canvas.width = window.innerWidth;
  canvas.height = window.innerHeight;

  class Particle {

  }

  class Effect { |<<<<<-----

  }

  function animate { |<<<<<-----

  }
})

```

Assim, a classe `Particle` cria individualmente cada partícula. `Effect` lida com todas as partículas criadas. e `animate` vai fazer com que as partículas tenham animação e sejam interativas.

Vamos criar um método para a nossa primeira classe, chamado `constructor()` esse método vai atribuir ao objeto novo valores e propriedades que vamos definir, quando chamarmos o objeto dessa forma: `Particle.constructor()`.

Vamos definir agora as propriedades da nossa partícula.

Cada partícula será um pedaço de uma imagem, um pixel, e todas as partículas combinadas criarão a imagem. Dessa forma ele vai precisar estar em uma posição muito específica, e precisamos das coordenadas de `x` e `y` para que o JavaScript saiba onde no `canvas` começar a desenhar.

Geralmente quando estamos trabalhando com desenvolvimento web, os navegadores tem as coordenadas `x` crescendo da esquerda para a direita, e `y` crescendo de cima para baixo. Se definirmos então que as coordenadas são 0 e 0 para os dois ele aparecerá no canto superior esquerdo.

```
class Particle {  
  constructor(){ |<<<<<-----  
    this.x = 0; |<<<<<-----  
    this.y = 0; |<<<<<-----  
  }  
}
```

Utilizamos o `this` para referenciar o objeto `particle` dentro dele mesmo. Assim o que estamos vendo, é a propriedade `x` nessa instância da classe `Particle`, que o `constructor` está criando agora. No fim o Escopo do `this` é o `Particle` e não a função que o cria.

Nossas partículas serão retângulos pois o `canvas` é mais rápido desenhando retângulos do que círculos. Vamos declarar agora o tamanho dos seus lados:

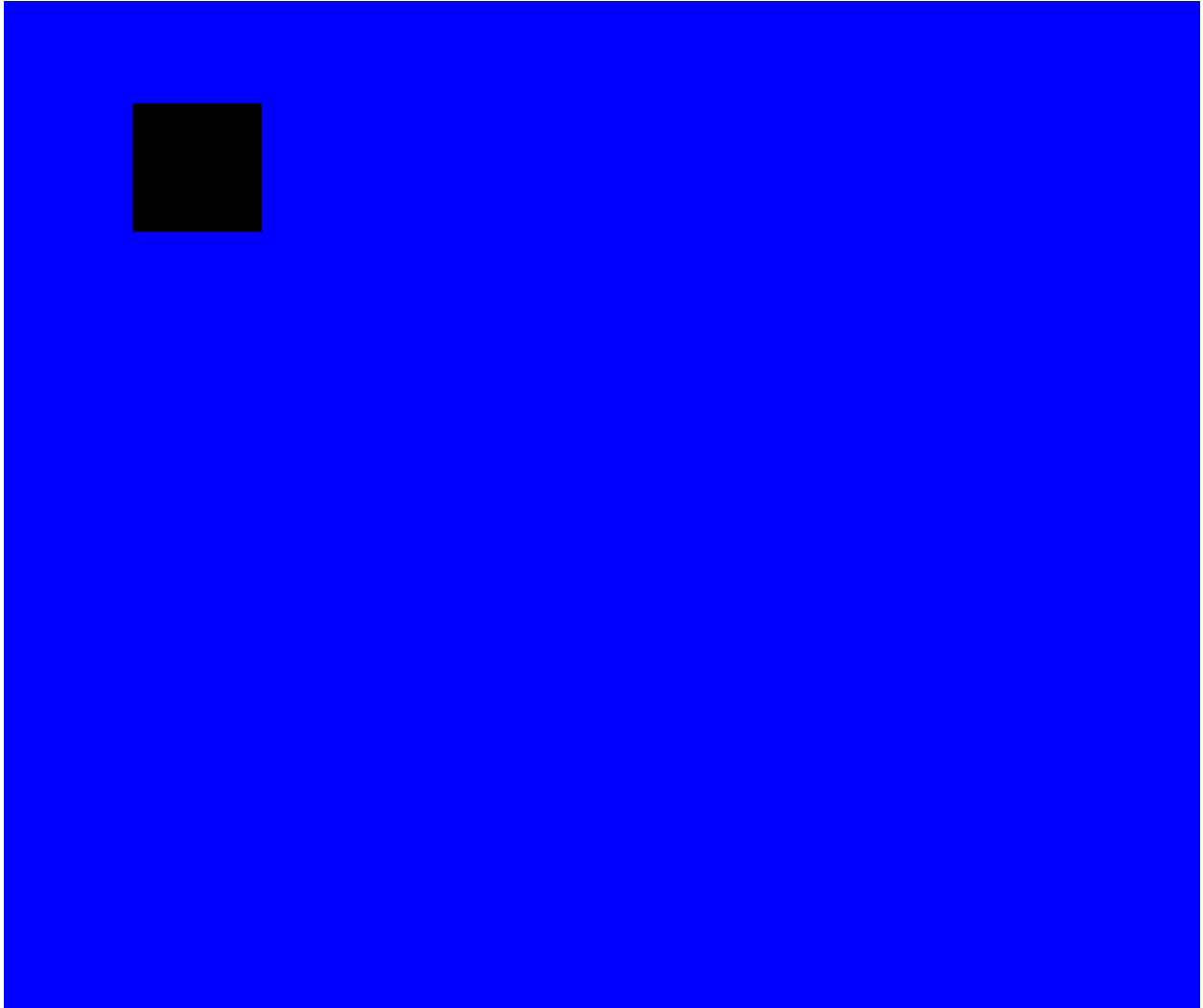
```
class Particle {  
  constructor(){  
    this.x = 0;  
    this.y = 0;  
    this.size = 3; |<<<<<-----  
  }  
}
```

5 DESENHANDO RETÂNGULOS

Para desenhar um retângulo no `canvas` tudo o que temos que fazer é chamar nossa constante `ctx` que guarda a instância do `canvas 2d drawing api` e chamarmos o método interno `fillRectangle`.

Por sua vez esse método espera 4 parâmetros, `(posição x, posição y, altura, largura)`.

```
ctx.fillRect(100, 100, 100, 100); |<<<<<-----
```



Se esse é o seu primeiro projeto trabalhando com `canvas` teste alterar esses valores para que você consiga visualizar o que está acontecendo.

Você consegue desenhar praticamente qualquer coisa com esses métodos: círculos, linhas, quadrados, losangos etc.

Hoje nós vamos nos aprofundar em imagens então vamos começar a trabalhar com elas.

6 DESENHANDO IMAGENS

Vamos começar selecionando o nosso elemento `img` do HTML através de uma constante.

```
window.addEventListener('load', function(){
  const canvas = document.getElementById('canvas1');
  const ctx = canvas.getContext('2d');
  canvas.width = window.innerWidth;
  canvas.height = window.innerHeight;

  const image1 = document.getElementById('image1') |<<<<<<<-----

  class Particle {
    constructor(){
      this.x = 0;
      this.y = 0;
      this.size = 3;
    }
  }

  class Effect {

  }

  function animate() {

  }

  ctx.fillRect(100, 100, 100, 100);
})
```

Logo depois vamos utilizar um método interno do nosso objeto `ctx` , para conseguirmos desenhar a imagem o `drawImage()` .

Esse método recebe 3 argumentos, **a imagem que queremos desenhar e as coordenadas x e y, para ele saber onde vai desenhar.**

assim:

```
const image1 = document.getElementById('image1');
ctx.drawImage(image1, 0, 0);
```

ela será desenhada no canto superior esquerdo, e se eu quiser mover ela 100 pixels para a direita:


```
const image1 = document.getElementById('image1');
ctx.drawImage(image1, 100, 0);
```

100 Pixels para baixo:

```
const image1 = document.getElementById('image1');
ctx.drawImage(image1, 0, 100);
```

Ainda nesse mesmo método existem 2 argumentos adicionais opcionais, que são a altura e a largura. Ou seja, se não passarmos esses argumentos o Java Script vai renderizar a imagem na tela em seu tamanho original.

Suponha então que você queira alterar as dimensões da imagem, o primeiro é a largura e o segundo é a altura, assim:

```
const image1 = document.getElementById('image1');
ctx.drawImage(image1, 0, 0, 100, 100);
```

A imagem agora tem 100px de altura e 100px de largura, isso vai restringir a imagem à área designada podendo causar distorções. Além disso é muito importante ter em mente que para uma melhor performance, o redimensionamento de imagem não deve ser feito com código, para isso utilize seu próprio editor de imagem.

7 DESENHANDO OS OBJETOS DE PARTÍCULAS

Então agora que sabemos como utilizar o método `fillRect()` para desenhar retângulos, e como usar o método `drawImage()` para desenhar imagens no `canvas`, vamos utilizar nossa classe `Particle` para desenhar alguma coisa.

Vamos excluir essa parte do código que serviu apenas de exemplo:

```
window.addEventListener('load', function(){
  const canvas = document.getElementById('canvas1');
  const ctx = canvas.getContext('2d');
  canvas.width = window.innerWidth;
  canvas.height = window.innerHeight;

  const image1 = document.getElementById('image1');
```

[illegible]

Vamos supor que queremos alterar o tamanho da nossa partícula para 30 x 30 (30px de altura e 30px de largura). Para conseguir isso basta alterar o tamanho da propriedade `this.size = 30;`.

```
class Particle {
    constructor(){
        this.x = 0;
        this.y = 0;
        this.size = 30;
    }
}
```

Agora para conseguir desenhar, vamos criar um método interno novo na classe `Particle`, que será responsável por pegar as propriedades definidas pelo `constructor()`, e desenhar uma partícula com os valores definidos por ela.

[illegible]

Ou seja, ela vai pegar o `this.size`, `this.x`, `this.y` e desenhar a partícula com essas medidas.

Vamos começar devagar e depois vamos alterando algumas coisas, para que fique mais adequado à programação orientada à objetos.

Assim, para desenhar algo precisamos pegar a nossa variável `ctx` e chamar a função interna `fillRect()` passando os quatro parâmetros necessários:

```
class Particle {  
  constructor(){  
    this.x = 0;  
    this.y = 0;  
    this.size = 30;  
  }  
  draw(){  
    ctx.fillRect(this.x, this.y, this.size, this.size);  
  }  
}
```

Perceba que passamos duas vezes o tamanho do retângulo que será desenhado, um para a largura e outro para altura. Como queremos na verdade um quadrado, temos as mesmas medidas de altura e largura, por isso a repetição.

Assim temos definida a nossa classe `Particle`, tendo nossa “planta” da casa (o `constructor()`) e o nosso método interno que pega as propriedades definidas antes e as aplica desenhando um retângulo.

Mas como você pôde perceber, se salvar o código e recarregar a página nada foi desenhado ainda. Como fazemos para esse retângulo ser desenhado?

Para conseguirmos desenhar esse retângulo, vamos precisar criar uma instância dessa classe.

- nota: Em programação orientada a objetos, chama-se **instância** de uma classe, um objeto cujo comportamento e estado são definidos pela classe.
([https://pt.wikipedia.org/wiki/Instância_\(ciência_da_computação\)](https://pt.wikipedia.org/wiki/Instância_(ciência_da_computação)))

Para criar uma instância da nossa classe partícula vamos criar uma variável chamada `particle1`, que receberá `new Particle()`.

A palavra `new` (novo) é um comando especial no JavaScript. Esse comando procura uma classe com esse nome, e vai automaticamente executar o método `constructor()` que por sua vez criará um novo objeto em branco, e vai vincular à ele as propriedades definidas no seu interior.

```
window.addEventListener('load', function(){
  const canvas = document.getElementById('canvas1');
  const ctx = canvas.getContext('2d');
  canvas.width = window.innerWidth;
  canvas.height = window.innerHeight;

  const image1 = document.getElementById('image1');

  class Particle {
    constructor(){
      this.x = 0;
      this.y = 0;
      this.size = 30;
    }
    draw(){
      ctx.fillRect(this.x, this.y, this.size, this.size);
    }
  }

  class Effect {

  }

  const particle1 = new Particle(); |<<<<<<<<-----

  function animate() {

  }

})
```

E agora que a nossa instância está criada, podemos ter acesso à todos os métodos internos à ele.

Chamaremos o nossos método `draw()`.

```
window.addEventListener('load', function(){
  const canvas = document.getElementById('canvas1');
  const ctx = canvas.getContext('2d');
  canvas.width = window.innerWidth;
  canvas.height = window.innerHeight;
```

```

const image1 = document.getElementById('image1');

class Particle {
  constructor(){
    this.x = 0;
    this.y = 0;
    this.size = 30;
  }
  draw(){
    ctx.fillRect(this.x, this.y, this.size, this.size);
  }
}

class Effect {

}

const particle1 = new Particle();
particle1.draw();      |<<<<<<<<-----

function animate() {

}

})

```

Assim, estamos usando nossa classe para criar e desenhar um objeto.

Agora que temos nosso quadrado desenhado, teste um pouco, alterando os valores das propriedades definidas no `constructor()`.

8 A CLASSE DE EFEITO

Para criar mais de uma partícula podemos replicar a criação da instância várias vezes.

Porém, isso não é eficiente e muito menos elegante. É sempre bom seu código ser *dry* (acrônimo para *don't repeat yourself*, que por sua vez significa não repita a si mesmo ou não repita código, em português podemos traduzir *dry* como **seco**, ou em melhor localização **enxuto**).

Vamos nesse tutorial manter tudo de acordo com orientação à objetos. Assim, vamos usar nossa classe que criamos `Effect` para criar e lidar com várias partículas ao mesmo tempo.

Assim, a classe `Particle` vai cuidar das partículas individuais, e a classe `Effect` vai lidar com todas ao mesmo tempo.

Começaremos então com o `constructor()` mas dessa vez ele vai esperar dois argumentos: `height` (altura) e `width` (largura). Vamos fazer isso, pois assim podemos garantir que a nossa classe `Effect` tem as dimensões atuais do nosso `canvas`.

```
class Effect {
  constructor(width, height){ <<<<<<<<<-----
  }
}
```

Agora vamos transformar esses parâmetros em propriedades da nossa classe:

```
class Effect {  
    constructor(width, height){  
        this.width = width;      |<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<  
        this.height = height;   |  
    }  
}
```

O que fizemos foi: os valores recebidos pelos parâmetros, serão guardados nas propriedades da classe das quais acabamos de criar. Ou seja, convertemos os parâmetros em propriedades.

Vamos ter mais uma propriedade chamada `particleArray` que vai receber todos os objetos criados pela classe `Particle`.

[illegible]

Vamos criar um método para a nossa classe `Effect` que vai se chamar `init()`. Esse método terá a função de iniciar o efeito e preencher o `array` que criamos anteriormente.

```
class Effect {
  constructor(width, height){
    this.width = width;
    this.height = height;
  }
}
```

```
        this.particlesArray = [];
    }
    init(){ |<<<<<<<<<<<<-----

    }
}
```

Vamos começar com apenas uma partícula, pegamos esse *array* e utilizamos o método `push()` do próprio Java Script que adiciona um ou mais elementos ao fim do *array*.

[illegible]

Se dermos como parâmetro uma instância nova da classe `Particle` para o `push`:

[illegible]

O que vai acontecer vai ser o seguinte: O método `push()` vai acionar a classe `Effect` e o método `constructor()` criando um objeto novo, no nosso caso uma partícula nova, baseado no modelo existente dentro dele.

Vamos deletar a criação das partículas do lado de fora da nossa classe `Effect` pois vamos criá-las dentro dela e não fora.

```
window.addEventListener('load', function(){
    const canvas = document.getElementById('canvas1');
```



```

class Effect {
  constructor(width, height){
    this.width = width;
    this.height = height;
    this.particlesArray = [];
  }
  init(){
    this.particlesArray.push(new Particle());
  }
  draw(){
    this.particlesArray.forEach(particle => particle.draw())
  }
}

```

`particle` é o nome temporário de cada um dos elementos do *array*, que pode ser qualquer nome.

Em seguida estamos utilizando uma arrow function que diz o que será aplicado a cada um dos elementos do array que chamamos de `particle`.

por exemplo:

```

function soma(n){
  n = n + 2;
}
numeros = [1, 2, 3, 4, 5, 6]
array.forEach(numero => soma(numero))

```

o que estamos fazendo é:

```

numeros = [numero[0] + 2, numero[1] + 2, numero[2] + 2, numero[3] + 2, numero[4] + 2, numero[5] + 2];

```

O método `draw()` que estamos chamando na verdade é o que está presente na classe `Particle`.

```

window.addEventListener('load', function(){
  const canvas = document.getElementById('canvas1');
  const ctx = canvas.getContext('2d');
  canvas.width = window.innerWidth;
  canvas.height = window.innerHeight;

```

```

const image1 = document.getElementById('image1');

class Particle {
  constructor(){
    this.x = 0;
    this.y = 0;
    this.size = 30;
  }
  draw(){
    ctx.fillRect(this.x, this.y, this.size, this.size); |<<<<<<-----
  }
}

class Effect {
  constructor(width, height){
    this.width = width;
    this.height = height;
    this.particlesArray = [];
  }
  init(){
    this.particlesArray.push(new Particle());
  }
  draw(){
    this.particlesArray.forEach(particle => particle.draw()); |<<<<<<-----
  }
}

function animate() {

}

})

```

Entendemos então que, o que existe dentro do nosso *array*, é o objeto que criamos a partir dessa primeira classe, assim podemos acessar seus métodos. Dessa forma, o `particle` passado como parâmetro é o objeto criado com base na classe `Particle` e assim podemos chamar seus métodos.

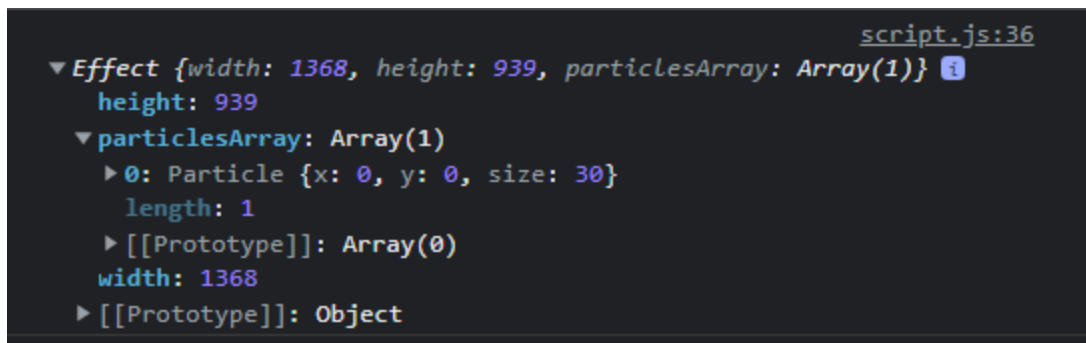
Agora vamos criar uma variável, e criar uma nova instância baseada na classe `Effect`. Como sabemos que o método `constructor` dessa classe precisa de dois parâmetros vamos passar para Ele.

Nesse caso os valores serão a largura e altura do `canvas`.

Depois de criado, vamos agora adicionar iniciar a criação de uma nova instância baseada na classe `Particle` com o método `init()` da nossa classe `Effect`.

```
const effect = new Effect(canvas.width, canvas.height);
effect.init();
console.log(effect);
```

Por fim damos um `log` só para ver se está tudo certo. e depois excluímos.



Está tudo certo com o nosso `array`, e com as propriedades de altura e largura que tem exatamente o tamanho disponível da tela. Se quiser testar se esses tamanhos se alteram, diminua a largura ou a altura da janela do navegador e recarregue a página, você observará os tamanho se alterando.

Observe também que as propriedades do objeto do `array` estão corretas. É sempre interessante testar de tempo em tempo se os valores estão certos.

Agora se chamarmos o método `draw()` do `Effect` vamos ter nossa partícula desenhada.

9 LIMPANDO O CÓDIGO

Ao analisar nosso código, percebemos que estamos fazendo algo que podemos melhorar: estamos chamando nosso `ctx` diretamente de dentro da nossa classe `Particle` isso não é uma boa prática.

```
const canvas = document.getElementById('canvas1');
const ctx = canvas.getContext('2d'); |<<<<<<<-----
canvas.width = window.innerWidth;
canvas.height = window.innerHeight;
```

```

const image1 = document.getElementById('image1');

class Particle {
  constructor(){
    this.x = 0;
    this.y = 0;
    this.size = 30;
  }
  draw(){
    ctx.fillRect(this.x, this.y, this.size, this.size); |<<<<<<-----
  }
}

```

Como queremos que o nosso código seja completamente modular ou pelo menos o máximo possível, vamos passar o `ctx` como parâmetro, ao invés de chamar ele diretamente.

Também vamos precisar fazer alterações passando esse mesmo parâmetro por outros locais:

```

window.addEventListener('load', function(){
  const canvas = document.getElementById('canvas1');
  const ctx = canvas.getContext('2d');
  canvas.width = window.innerWidth;
  canvas.height = window.innerHeight;

  const image1 = document.getElementById('image1');

  class Particle {
    constructor(){
      this.x = 0;
      this.y = 0;
      this.size = 30;
    }
    draw(context){ |<<<<<<-----
      context.fillRect(this.x, this.y, this.size, this.size); |<<<<<<-----
    }
  }

  class Effect {
    constructor(width, height){
      this.width = width;
      this.height = height;
      this.particlesArray = [];
    }
    init(){
      this.particlesArray.push(new Particle());
    }
    draw(context){ |<<<<<<-----

```

```

        this.particlesArray.forEach(particle => particle.draw(context));|<<<<<<----
    }
}

const effect = new Effect(canvas.width, canvas.height);
effect.init();
effect.draw(ctx); |<<<<<<-----

function animate() {

}

})

```

Tivemos que passar esse parâmetro por dentro da nossa classe `Effect` pois é ela quem vai chamar o método da classe `Particle`.

10 VÁRIAS PARTÍCULAS ALEATÓRIAS

Podemos gerar um posicionamento aleatório para as partículas que são geradas no nosso `canvas`. Para fazer isso basta trocar o valor de `x` e `y` na nossa classe `Particle`, para `Math.random() * canvas.width;` para `x`, e `Math.random() * canvas.height;` para `y`. Isso vai resultar em um número aleatório entre 0 e o tamanho da largura da página e a altura da página para seus respectivas propriedades.

```

class Particle {
  constructor(){
    this.x = Math.random() * canvas.width; |<<<<<<-----
    this.y = Math.random() * canvas.height; |<<<<<<-----
    this.size = 30;
  }
  draw(context){
    context.fillRect(this.x, this.y, this.size, this.size);
  }
}

```

Mas novamente estamos fazendo nosso código depender demais do código externo, quando utilizamos o `canvas.width` e `canvas.height`.

Mas como vamos fazer então para que cada partícula saiba o tamanho do `canvas`, e gerar a posição aleatória? Podemos passar para o `constructor()` da `Particle` a classe `Effect` inteira, pois como sabemos, ela guarda o tamanho da tela. Logo depois, vamos

converter dentro do próprio `constructor` a classe `Effect` passada como uma propriedade.

mas isso não resultaria em uma cópia da classe inteira cada vez que eu criasse uma nova instância da `Particle`? No Java Script objetos são chamados também de `reference data types`. O que significa que ao passar um parâmetro, eu não estou salvando uma cópia toda vez que eu crio uma partícula, eu estou salvando o “endereço”. É uma referência para o local na memória de onde aquela informação está. Assim, eu não crio uma cópia toda vez que eu crio um objeto novo.

Agora que temos essa referência dentro da nossa primeira classe, podemos acessar qualquer propriedade de dentro dela, e por isso eu posso utilizar `this.effect.width` e `this.effect.height`:

```
class Particle {
  constructor(effect){ |<<<<<<-----
    this.effect = effect; |<<<<<<-----
    this.x = Math.random() * this.effect.width; |<<<<<<-----
    this.y = Math.random() * this.effect.height; |<<<<<<-----
    this.size = 30;
  }
  draw(context){
    context.fillRect(this.x, this.y, this.size, this.size);
  }
}

class Effect {
  constructor(width, height){
    this.width = width;
    this.height = height;
    this.particlesArray = [];
  }
  init(){
    this.particlesArray.push(new Particle(this)); |<<<<<<-----
  }
  draw(context){
    this.particlesArray.forEach(particle => particle.draw(context));
  }
}
```

Não podemos esquecer que temos que passar a classe como parâmetro e para isso utilizamos o `this`.

para aumentar o número de partículas vamos simplesmente criar um loop for com 10 repetições:

```

class Effect {
  constructor(width, height){
    this.width = width;
    this.height = height;
    this.particlesArray = [];
  }
  init(){
    for(let i = 0; i < 10; i++){
      this.particlesArray.push(new Particle(this));
    }
  }
  draw(context){
    this.particlesArray.forEach(particle => particle.draw(context));
  }
}

```

11 O MÉTODO `drawImage()`

Vamos desenhar a nossa imagem. Porém como vimos antes nossa imagem está sendo selecionada do arquivo HTML por uma variável que está do lado de fora das nossas classes e o método que vai desenhar a imagem vai ficar dentro delas, assim, nós não queremos ficar dependendo do código externo.

Para resolver esse problema vamos simplesmente tornar essa imagem em uma propriedade da classe `Effect`.

```

class Effect {
  constructor(width, height){
    this.width = width;
    this.height = height;
    this.particlesArray = [];
    this.image = document.getElementById('image1');
  }
  init(){
    for(let i = 0; i < 10; i++){
      this.particlesArray.push(new Particle(this));
    }
  }
  draw(context){
    this.particlesArray.forEach(particle => particle.draw(context));
  }
}

```

Como transformamos a variável em uma propriedade vamos simplesmente trocar a declaração `const` para `this` referindo-se que esse valor é relacionado à classe `Effect`.

OBS: o nome da variável era `image1`, vamos manter `image`.

Também podemos desenhar essa imagem chamando o método `drawImage()` logo abaixo no método `draw()`.

```
draw(context){
  this.particlesArray.forEach(particle => particle.draw(context));
  context.drawImage(this.image, 0, 0); |<<<<<<-----
}
```

Fazemos dessa forma para conseguirmos seguir o padrão do paradigma de orientação à objetos.

12 COMO CENTRALIZAR IMAGENS NO CANVAS

Imagens no elemento `canvas` são desenhadas começando do canto superior esquerdo e sendo geradas da esquerda para a direita e de cima para baixo. Vamos aprender agora como centralizar essas imagens.

Para isso vamos criar duas propriedades novas na nossa classe `Effect`:

```
class Effect {
  constructor(width, height){
    this.width = width;
    this.height = height;
    this.particlesArray = [];
    this.image = document.getElementById('image1');
    this.centerX = this.width * 0.5; |<<<<<<-----
    this.centerY = this.width * 0.5; |<<<<<<-----
  }
  init(){
    for(let i = 0; i < 10; i++){
      this.particlesArray.push(new Particle(this));
    }
  }
  draw(context){
    this.particlesArray.forEach(particle => particle.draw(context));
    context.drawImage(this.image, 0, 0);
  }
}
```


Essas propriedades possuem o valor da largura da tela e da altura, e ambas são multiplicadas por `0.5`, para que consigamos a metade do valor da tela.

Agora podemos utilizar essas duas novas propriedades para passar como parâmetro e escolher onde a imagem será desenhada.

Mas ainda vemos que a imagem não está centralizada, e mal aparece na tela. Isso acontece pois o elemento `` possui valores próprios de de altura e largura. Podemos ver isso mais claramente se imprimirmos na tela seus valores. teste ai:

```
class Effect {
  constructor(width, height){
    this.width = width;
    this.height = height;
    this.particlesArray = [];
    this.image = document.getElementById('image1');
    this.centerX = this.width * 0.5;
    this.centerY = this.width * 0.5;
    console.log(this.image.width); |<<<<<<-----
    console.log(this.image.height); |<<<<<<-----
  }
  ...
}
```

Para resolver esse conflito vamos mexer nesses valores também, criando duas propriedades novas:

```
class Effect {
  constructor(width, height){
    this.width = width;
    this.height = height;
    this.particlesArray = [];
    this.image = document.getElementById('image1');
    this.centerX = this.width * 0.5;
    this.centerY = this.height * 0.5;
    this.x = this.centerX - this.image.width * 0.5; |<<<<<<-----
    this.y = this.centerY - this.image.height * 0.5; |<<<<<<-----
  }
  ...
}
```

O que fizemos é: achamos os valores do centro da tela com o `this.centerX` e o `this.centerY` e depois diminuimos desse valor o tamanho da imagem pela metade.

E agora está centralizado, pois o ponto central que achamos deixou de estar no centro e deu lugar à imagem.

13 MOVIMENTAÇÃO DAS PARTÍCULAS

Até agora estamos desenhando nossa imagem e 10 partículas.

Vamos alterar o tamanho das partículas para 5 e também a quantidade para 100;

```
class Particle {
  constructor(effect){
    this.effect = effect;
    this.x = Math.random() * this.effect.width;
    this.y = Math.random() * this.effect.height;
    this.size = 5; |<<<<<<<-----
  }
  draw(context){
    context.fillRect(this.x, this.y, this.size, this.size);
  }
}

class Effect {
  constructor(width, height){
    this.width = width;
    this.height = height;
    this.particlesArray = [];
    this.image = document.getElementById('image1');
    this.centerX = this.width * 0.5;
    this.centerY = this.height * 0.5;
    this.x = this.centerX - this.image.width * 0.5;
    this.y = this.centerY - this.image.height * 0.5;
  }
  init(){
    for(let i = 0; i < 100; i++){ |<<<<<<<-----
      this.particlesArray.push(new Particle(this));
    }
  }
  draw(context){
    this.particlesArray.forEach(particle => particle.draw(context));
    context.drawImage(this.image, this.x, this.y);
  }
}
```

Nossas partículas, toda vez que recarregamos a página, assumem um lugar diferente e aleatório na tela, de acordo com o que fizemos até agora.

Também podemos fazer o tamanho das nossas partículas serem aleatórias, como também já sabemos, assim como também podemos fazer elas se moverem.

Para fazer isso vamos começar criando uma nova propriedade na classe `Particle` que receberá o nome de `update()`, e uma propriedade que receberá o nome de `vx` (velocity x), que guardará o valor da velocidade no eixo x.

O valor será 1. Isso significa que as partículas vão se movimentar 1 pixel por frame de animação.

A mesma coisa vai funcionar para a propriedade `vy`, que será nossa velocidade vertical, ou seja, no eixo y.

```
class Particle {
  constructor(effect){
    this.effect = effect;
    this.x = Math.random() * this.effect.width;
    this.y = Math.random() * this.effect.height;
    this.size = 5;
    this.vx = 1; |<<<<<<<-----
    this.vy = 1; |<<<<<<<-----
  }
  draw(context){
    context.fillRect(this.x, this.y, this.size, this.size);
  }
  update(){ |<<<<<<<-----
}
}
```

Dentro do método `update()` vamos descrever o que vai acontecer com cada partícula em cada frame de animação, que por sua vez, será o acréscimo da sua posição em 1.

Assim, em cada frame de animação o valor definido como a velocidade no eixo x e y serão adicionados ao valor de altura e largura que as partículas possuem atualmente.

```
update(){
  this.x += this.vx; |<<<<<<<-----
  this.y += this.vy; |<<<<<<<-----
}
```

Assim cada partícula terá o seu próprio método `update()` que irá definir seu movimento, ou em outras palavras, sua posição em cada frame de animação.

Também vamos criar um método `update()` só que agora na classe `Effect`. O trabalho deste aqui, será chamar o `update()` da classe `Particle` para cada partícula existente

dentro do *array* de partículas.

```
class Effect {
  constructor(width, height){
    this.width = width;
    this.height = height;
    this.particlesArray = [];
    this.image = document.getElementById('image1');
    this.centerX = this.width * 0.5;
    this.centerY = this.height * 0.5;
    this.x = this.centerX - this.image.width * 0.5;
    this.y = this.centerY - this.image.height * 0.5;
  }
  init(){
    for(let i = 0; i < 100; i++){
      this.particlesArray.push(new Particle(this));
    }
  }
  draw(context){
    this.particlesArray.forEach(particle => particle.draw(context));
    context.drawImage(this.image, this.x, this.y);
  }
  update(){
    this.particlesArray.forEach(particle => particle.update()); |<<<<-----
  }
}
```

Agora vamos adicionar esse método logo abaixo onde chamamos os outros:

```
effect.init();
effect.draw(ctx);
effect.update(); |<<<<-----
```

Mas nada aparentemente aconteceu.

O motivo disso é que, uma vez carregadas, as partículas não são geradas novamente.

Como definimos antes, os métodos `update's()` vão definir uma posição nova a cada frame de animação, e como não temos nenhum, não vemos o efeito que queremos. Assim, vamos precisar criar um loop de animação com os frames necessários para a mudança de posição.

Para fazer esse loop, vamos precisar criar uma nova função `animate`.

```

effect.init();

animate(){
    effect.draw(ctx);
    effect.update();
}

```

Essa função terá `effect.draw` e o `effect.update` dentro delas, assim podemos retirar os dois do *flow* normal e colocar dentro dessa função.

Dentro dessa função ainda vamos utilizar uma função nativa do objeto *window*, mas que podemos chamar diretamente:

```

animate(){
    effect.draw(ctx);
    effect.update();
    requestAnimationFrame();
}

```

Esse método vai chamar uma função que passarmos para ela como argumento antes do próximo `repaint` assim, se dermos o nome da sua função pai como parâmetro ela irá criar um loop, criando partículas e as desenhando de novo e de novo criando a animação.

```

animate(){
    effect.draw(ctx);
    effect.update();
    requestAnimationFrame(animate);
}

```

O número de vezes que esse método chama a função passada como parâmetro é normalmente 60 vezes por segundo, gerando assim uma animação de 60 frames por segundo, mas pode variar já que ele vai tentar igualar a quantidade de vezes com a taxa de atualização do seu monitor.

Na maioria dos navegadores modernos, a animação será pausada quando estiver em uma aba em segundo plano para melhorar a performance e salvar a bateria do seu notebook.

Agora, quando recarregamos a página podemos ver que a animação cria um rastro e não parece que as partículas estão se movendo. Para fazer isso precisamos limpar o `canvas` toda vez que as partículas forem redesenhadas. Fazemos isso com o método nativo do do nosso `ctx`, `clearRect()`.

```
function animate(){
  ctx.clearRect(0, 0, canvas.width, canvas.height); |<<<<-----
  effect.draw(ctx);
  effect.update();
  requestAnimationFrame(animate);
}
```

Os parâmetros estão dizendo ao método que queremos limpar do ponto 0 no eixo `x` e `y`, até o ponto máximo da tela, no eixo `x` e `y`.

Como as partículas estão se movendo por conta da velocidade que definimos, podemos também colocar diferentes velocidades ou até mesmo velocidades aleatórias.

Ainda, se quisermos que as nossas partículas possam se mover em todas as direções possíveis, podemos fazer com que os valores comecem negativos, as partículas que receberem velocidade negativa no eixo `x` por exemplo ficam se movendo para esquerda e as que receberem velocidade positiva para a direita.

já no eixo `y` as partículas que recebem valores negativos sobem e as que recebem valores positivos descem.

```
class Particle {
  constructor(effect){
    this.effect = effect;
    this.x = Math.random() * this.effect.width;
    this.y = Math.random() * this.effect.height;
    this.size = 5;
    this.vx = Math.random() * 2 - 1; |<<<<-----
    this.vy = Math.random() * 2 - 1; |<<<<-----
  }
  draw(context){
    context.fillRect(this.x, this.y, this.size, this.size);
  }
  update(){
    this.x += this.vx;
    this.y += this.vy;
  }
}
```

Vamos simplesmente retirar o fundo azul do `canvas` no CSS e pronto.

Agora, como vamos fazer para que essas partículas assumam o formato da imagem? como fazer para que elas se lembrem desse formato e depois que eu quebrar a imagem elas retornem para o ponto que queremos?

Até agora cobrimos o básico sobre o `canvas`. A partir daqui entraremos em um território um pouco mais avançado.

Se você chegou até aqui e é um iniciante, meus parabéns! a próxima parte pode ser um pouco mais desafiadora, então não desanime! tentarei te ajudar o máximo possível, sendo o mais claro que eu puder na explicação.

14 PIXEL ANALYSIS COM `getImageData()`

Nossa classe `Particle` tem a função de criar uma partícula nova a cada vez que é chamada. Atualmente nossas partículas são simples quadrados pretos.

Na verdade o que queremos é “*pixelar*” a imagem e fazer cada bloco que criamos ter uma cor específica.

A partícula também precisa lembrar o seu ponto inicial, visto que queremos movê-las pela página e fazer algumas outras coisinhas com elas. Para isso então, precisamos que elas se lembrem do seu ponto inicial, achando seu caminho de volta até ele recriando assim a imagem.

Na nossa classe `Effect` temos o método `init()` que cria 100 objetos (número definido no loop) e os coloca dentro do *array*. Em outras palavras, esse método cria 100 partículas e adiciona elas na nossa coleção de partículas.

Vamos precisar de outra forma para conseguir quebrar nossa imagem em partículas, Assim vamos reescrever o método `init()`:

```
class Effect {
  constructor(width, height){
    this.width = width;
    this.height = height;
    this.particlesArray = [];
    this.image = document.getElementById('image1');
    this.centerX = this.width * 0.5;
    this.centerY = this.height * 0.5;
    this.x = this.centerX - this.image.width * 0.5;
    this.y = this.centerY - this.image.height * 0.5;
  }
}
```

```

init(){
    |<<<<-----
}
draw(context){
    this.particlesArray.forEach(particle => particle.draw(context));
    context.drawImage(this.image, this.x, this.y);
}
update(){
    this.particlesArray.forEach(particle => particle.update());
}
}

```

Lembre-se que as propriedades `this.x` e `this.y` definem onde a imagem será desenhada. Então se colocarmos por exemplo no método `update()` da classe `Effect`, `this.x++`, a imagem irá se mover para a direita pois a cada chamada da função estamos adicionando 1 ao valor da propriedade `this.x`.

Não esqueça esses atributos, eles serão importante lá na frente.

Agora vamos deletar do método `draw()` a função de desenhar a imagem e passar para o método `init()`:

```

init(){
    context.drawImage(this.image, this.x, this.y); |<<<<-----
}
draw(context){
    this.particlesArray.forEach(particle => particle.draw(context));
}

```

O trabalho do `init()` agora vai ser desenhar a imagem no `canvas`, analisá-la e quebrá-la em partículas. Depois podemos deletar a imagem do `canvas` pois as nossas partículas vão se tornar a imagem.

Como estamos desenhando agora no método `init()` precisamos passar o nosso contexto para ele como parâmetro:

```

init(context){ |<<<<-----
    context.drawImage(this.image, this.x, this.y);
}
draw(context){
    this.particlesArray.forEach(particle => particle.draw(context));
}

```


Assim como também chamamos o método `init()` lá na frente precisamos passar então o nosso `ctx` como parâmetro para ele:

```
effect.init(ctx); |<<<<-----
```

Esse método vai rodar apenas uma vez quando a página for carregada e vai em seguida deletar imediatamente a imagem após ser desenhada no `canvas`, pois a nossa função `animate()` vai limpar o `canvas` logo depois.

Por enquanto vamos tirar a chamada da função `animate()` apenas comentando sua linha.

Agora que conseguimos ver nossa imagem, vamos focar no `init()`.

Se voltarmos um pouco na nossa classe `Particle` sabemos que ela cria uma partícula com as propriedades que definimos, e sabemos que elas se movem pois seu valor foi atualizado conforme também estabelecemos.

Como também sabemos ela cria quadrados pretos, e não é bem isso que queremos.

Vamos transformar cada pixel da imagem em uma partícula.

Dentro do nosso método `init()` criaremos uma variável temporária chamada `pixels`. Ela será um *array* que guardará todas as posições e valores de cores de cada pixel no `canvas`.

```
init(context){  
  context.drawImage(this.image, this.x, this.y);  
  const pixels = ; |<<<<-----  
}
```

Para fazer a análise da imagem, nós primeiro precisamos desenhá-la no `canvas` (como fizemos na linha anterior), e só depois poderemos obter seus dados.

Assim, depois de desenhada a imagem, podemos utilizar nosso método `getImageData()` que analisa um pedaço específico do nosso `canvas` e nos devolve suas informações (*pixel data*) em um formato de objeto. É por isso que precisamos desenhar a imagem primeiro, pois esse método analisa o `canvas` e não a imagem.

Como vamos utilizar um método que vai trabalhar com o nosso `canvas` ele é um método nativo do nosso `ctx`, e sabemos que ele está sendo passado para dentro do

nosso método como `context`.

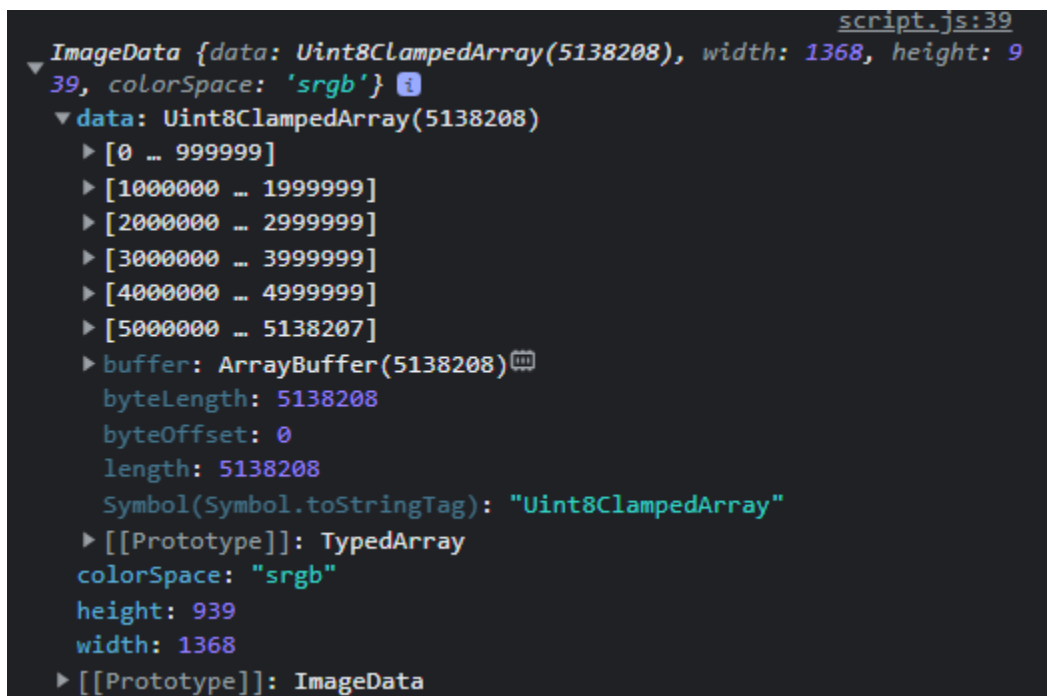
```
init(context){
  context.drawImage(this.image, this.x, this.y);
  const pixels = context.getImageData(0, 0, this.width, this.height ); |<<<<-----
}
```

Como o nosso método trabalha com o pedaço do `canvas` ele recebe quatro argumentos que dizem para ele de onde até onde ele precisa trabalhar.

Aqui vale o aviso: se você tentar utilizar o `getImageData()` com uma imagem em um arquivo que não faça parte do seu HTML, como por exemplo um arquivo `.png`, você receberá um erro informando que você tem arquivos com conflito de origem (*cross-origin error*). Isso acontecerá mesmo que você tenha o arquivo na mesma pasta do arquivo principal do html.

É por esse motivo que passamos a imagem para formato `base64` e tornamos ela parte da HTML ao invés da HTML indicar onde está a imagem.

Dê um console log para ver as informações obtidas.



Podemos ver que o nosso processo deu o retorno esperado. Dentro dele temos um *array* gigantesco que trás informações sobre cada um dos pixels da porção do `canvas`

que estabelecemos, que no nosso caso é o `canvas` inteiro.

Esse *array* é de um tipo especial chamado `unit8 clamped array` que basicamente é um *array* feito de `unsigned 8-bit integers` (*unsigned integers* são comumente chamados de *unit*, como são no nosso caso de até 8bits, daí o nome do *array*). *Unsigned integers* (inteiros não assinalados) são números inteiros não assinalados como positivos ou negativos, por isso o nome.

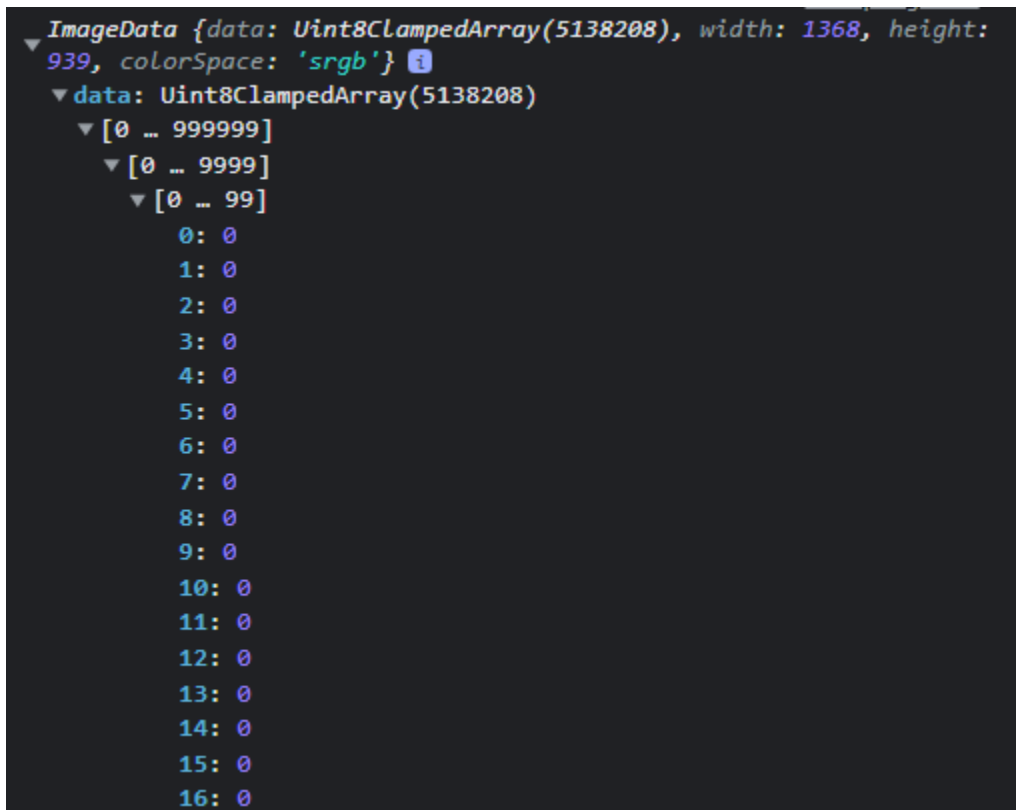
O `clamped` (limitado) significa que os números possuem um limite de 0 até 255.

Se você já trabalhou antes com cores no desenvolvimento web, sabe que todas as cores podem ser expressas pela declaração tipo RGB e alpha (camada de opacidade), ou RGBA. Utilizamos em CSS o tempo inteiro.

Nesse caso, *Red* (vermelho), *Green* (verde) e *Blue* (azul) podem receber qualquer valor de 0 até 255, e o canal alpha recebe um valor de 0 até 1, sendo 0 completamente transparente e 1 completamente visível.

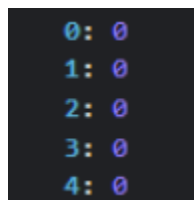
O nosso *array* é dividido de uma maneira em que a cada quatro elementos temos a representação de 1 pixel. Ou seja, a cada quatro elementos temos a representação do RGB e A de cada pixel. essa informação será importante, então guarde: 4 elementos nesse *array* representam 1 pixel.

Vamos abrir o início do nosso *array*:



O nosso `canvas` começa a ser analisado pelo ponto que definimos como sendo o inicial que é o `0, 0`. isso significa que o ponto inicial é o canto superior esquerdo.

se analisarmos os quatro primeiros valores vamos ver que esse pixel é um pixel preto transparente pois possui nos três primeiros valores 0, e na camada de opacidade também 0.



podemos perceber que esses valores se repetem até que valores diferentes começam a aparecer, que será onde nossa imagem estará desenhada.

Se você procurar pelo meio do `array`, vai começar a perceber a aparição desses valores.

15 EXTRAINDO COORDENADAS E CORES DO NOSSO PIXEL DATA

Delete o `console.log`, não vamos precisar mais dele.

Nossa variável `pixels` agora guarda um objeto com as informações do pedaço do `canvas` que selecionamos.

Como não queremos que ele guarde o objeto inteiro e só o *array*, vamos acessar apenas o array:

```
init(context){
  context.drawImage(this.image, this.x, this.y);
  const pixels = context.getImageData(0, 0, this.width, this.height).data; |<<<<-----
}
```

Vamos vasculhar todo esse *array*, e quando encontrarmos um valor de transparência que seja diferente de 0, vamos criar um objeto, utilizando nossa classe `Particle` para cada pixel com essa característica, que o representará.

Para conseguirmos analisar linha por linha vamos precisar utilizar aninhamento de loops, que nada mais é do que um loop dentro de outro.

```
class Effect {
  constructor(width, height){
    this.width = width;
    this.height = height;
    this.particlesArray = [];
    this.image = document.getElementById('image1');
    this.centerX = this.width * 0.5;
    this.centerY = this.height * 0.5;
    this.x = this.centerX - this.image.width * 0.5;
    this.y = this.centerY - this.image.height * 0.5;
    this.gap = 5; |<<<<-----
  }
  init(context){
    context.drawImage(this.image, this.x, this.y);
    const pixels = context.getImageData(0, 0, this.width, this.height).data;
    for (let y = 0; y < this.height; y += this.gap){ |<<<<-----

      }
    ...
  }
}
```

Observe que se a atualização do nosso loop (`y += this.gap`) fosse 1, estaríamos definindo que a nossa linha teria 1 pixel. Ou seja, depois de analisar uma linha, nosso loop iria para a próxima.

Quando o loop terminasse, para cada píxel da imagem ele teria um novo objeto. O problema é que se criássemos um objeto para cada píxel o site ficaria muito pesado, e a animação não ficaria interessante. É possível fazer dessa forma, mas isso precisa de otimizações mais complexas e não vamos trabalhar isso nesse material.

Por isso criamos uma propriedade chamada `this.gap` que vai representar o tamanho da linha. ou seja, terminou de analisar uma linha ele vai pular 5 pixels e vai para a próxima, totalizando menos objetos criados, resultando em um site mais leve. Assim ele vai mais ou menos ter 5 pixels.

Não importa muito o valor que dermos agora (que no caso foi 5), poderemos alterar isso depois, mas o importante é saber que quanto maior o valor no gap, maior a performance mas a imagem fica mais “pixelada”.

Agora, vamos fazer a análise das linhas com o outro loop, que vai da esquerda para a direita:

```
init(context){
  context.drawImage(this.image, this.x, this.y);
  const pixels = context.getImageData(0, 0, this.width, this.height).data;
  for (let y = 0; y < this.height; y += this.gap){
    for(let x = 0; x < this.width; x += this.gap){

    }
  }
}
```

Primeiramente pode parecer confuso trabalhar com loops aninhados, mas observe que o loop de fora vai mapear e pular verticalmente, enquanto o de dentro vai mapear e analisar cada elemento da linha horizontalmente.

Assim, o que vai acontecer é o seguinte: começamos do primeiro loop, entramos no segundo, enquanto o valor de x for menor do que o do tamanho máximo do `canvas`, ele vai passar por todos os pixels dele, e ele vai fazer o que estamos pedindo dentro do loop. Quando essa condição for falsa ele vai sair do loop e voltar para o de fora, que vai pular cinco pixels para baixo, e começar de novo, até que sejam analisados todos os pixels do `canvas`.

Em resumo estamos passando por todo o nosso `canvas` de 5 em 5 pixels, toda vez que passarmos para a nossa próxima célula no nosso grid imaginário. Queremos testar se existe alguma cor dentro dessa célula ou se estamos passando pela parte transparente.

Se encontrarmos alguma ou algumas cores dentro da célula vamos criar uma partícula usando nossa classe `Particle`, fazendo essa mesma partícula se lembrar da sua posição `x` e `y`, ou seja, sua posição na imagem, e sua cor.

O desafio é que enquanto passamos por esse array enorme de índices que vão de 0 até centenas de milhares, precisamos achar uma forma de pegar esses números e pegar suas coordenadas `x` e `y`, além das suas cores.

Vamos começar selecionando o índice dos pixels. A fórmula é um pouco avançada então não se preocupe se você não entender de primeira.

```
init(context){
  context.drawImage(this.image, this.x, this.y);
  const pixels = context.getImageData(0, 0, this.width, this.height).data;
  for (let y = 0; y < this.height; y += this.gap){
    for(let x = 0; x < this.width; x += this.gap){
      const index = (y * this.width + x) * 4; |<<<<<<-----
    }
  }
}
```

Como não é o intuito explicar o sentido dessa formula, apenas saiba que ela existe, e que é interessante saber como usá-la. Não se preocupe no momento em saber exatamente como ela funciona, com o avançar dos seus estudos você vai entender melhor seu funcionamento se for de seu interesse.

Sabemos que cada pixel é representado por quatro números inteiros no nosso *array*. Assim, sabemos que o valor para *Red* é o primeiro, que o *Green* é o segundo e assim por diante.

tomemos como exemplo essa parte:

```
700300: 6
700301: 17
700302: 14
700303: 255
```

isso resultaria no `rgb(6, 17, 14)`.

Assim o primeiro será o índice que achamos no `index` depois o segundo `index + 1` e assim por diante:

```
init(context){
  context.drawImage(this.image, this.x, this.y);
  const pixels = context.getImageData(0, 0, this.width, this.height).data;
  for (let y = 0; y < this.height; y += this.gap){
    for(let x = 0; x < this.width; x += this.gap){
      const index = (y * this.width + x) * 4;
      const red = pixels[index];          |<<<<<<<-----
      const green = pixels[index + 1];    |<<<<<<<-----
      const blue = pixels[index + 2];     |<<<<<<<-----
      const alpha = pixels[index + 3];    |<<<<<<<-----
    }
  }
}
```

Agora que temos nossos valores apenas precisamos fazer ele se tornar uma notação RGB, concatenando seus valores em uma `string`:

```
init(context){
  context.drawImage(this.image, this.x, this.y);
  const pixels = context.getImageData(0, 0, this.width, this.height).data;
  for (let y = 0; y < this.height; y += this.gap){
    for(let x = 0; x < this.width; x += this.gap){
      const index = (y * this.width + x) * 4;
      const red = pixels[index];
      const green = pixels[index + 1];
      const blue = pixels[index + 2];
      const alpha = pixels[index + 3];
      const color = 'rgb(' + red + ',' + green + ',' + blue + ')'; |<<<<<<<-----
    }
  }
}
```

Porém não queremos criar uma partícula nova para pixel vasculhado, apenas para os que tiverem alguma cor. Dessa forma vamos criar uma estrutura condicional que vai adicionar uma partícula nova em caso de o alpha ser maior do que 0.

```
init(context){
  context.drawImage(this.image, this.x, this.y);
  const pixels = context.getImageData(0, 0, this.width, this.height).data;
```



```

    for (let y = 0; y < this.height; y += this.gap){
      for(let x = 0; x < this.width; x += this.gap){
        const index = (y * this.width + x) * 4;
        const red = pixels[index];
        const green = pixels[index + 1];
        const blue = pixels[index + 2];
        const alpha = pixels[index + 3];
        const color = 'rgb(' + red + ',' + green + ',' + blue + ')';

        if(alpha > 0){
          this.particlesArray.push(new Particle()); |<<<<<-----
        }
      }
    }
  }
}

```

Nossa classe `Particle` sabe apenas como criar partículas pretas e em posições diferentes. Vamos precisar ensinar à ela como receber e trabalhar esses novos dados que temos, e se moldar corretamente para formar uma imagem. Assim, vamos adicionar mais parâmetros ao nosso `constructor()` da classe `Particle`.

```

class Particle {
  constructor(effect, x, y, color){ |<<<<<<-----
    this.effect = effect;
    this.x = Math.random() * this.effect.width;
    this.y = Math.random() * this.effect.height;
    this.size = 5;
    this.vx = Math.random() * 2 - 1;
    this.vy = Math.random() * 2 - 1;
  }
  draw(context){
    context.fillRect(this.x, this.y, this.size, this.size);
  }
  update(){
    this.x += this.vx;
    this.y += this.vy;
  }
}

```

agora vamos transformar esses parâmetros em novas propriedades:

`originX` → vai lembrar onde essa partícula estava no eixo x, por isso ela recebe o `x` que vamos calcular nos nossos loops que fizemos anteriormente. Vamos também arredondar eles para baixo com o auxílio da função nativa `Math.floor()`, isso para impedir que o `canvas` receba valores quebrados de pixel e utilize qualquer *anti-aliasing* (*anti-serrilhamento*).

Obs: é sempre interessante utilizar o arredondamento para baixo quando estamos trabalhando com o `canvas` pensando na performance.

`originY` → parecido com o `originX`, só que no eixo `y`.

`color` → esse virá daquela variável `color` que já fizemos no nosso loop. Assim cada partícula vai continuar com a cor original da imagem.

```
class Particle {
  constructor(effect, x, y, color){
    this.effect = effect;
    this.x = Math.random() * this.effect.width;
    this.y = Math.random() * this.effect.height;
    this.originX = Math.floor(x); |<<<<<<-----
    this.originY = Math.floor(y); |<<<<<<-----
    this.color = color;           |<<<<<<-----
    this.size = 5;
    this.vx = Math.random() * 2 - 1;
    this.vy = Math.random() * 2 - 1;
  }
  draw(context){
    context.fillRect(this.x, this.y, this.size, this.size);
  }
  update(){
    this.x += this.vx;
    this.y += this.vy;
  }
}
```

Agora o `constructor()` da classe `Particle` espera 4 argumentos. e precisamos passá-los na criação das nossas partículas.

```
class Effect {
  constructor(width, height){
    this.width = width;
    this.height = height;
    this.particlesArray = [];
    this.image = document.getElementById('image1');
    this.centerX = this.width * 0.5;
    this.centerY = this.height * 0.5;
    this.x = this.centerX - this.image.width * 0.5;
    this.y = this.centerY - this.image.height * 0.5;
    this.gap = 5;
  }
  init(context){
    context.drawImage(this.image, this.x, this.y);
    const pixels = context.getImageData(0, 0, this.width, this.height).data;
    for (let y = 0; y < this.height; y += this.gap){
```

```

        for(let x = 0; x < this.width; x += this.gap){
            const index = (y * this.width + x) * 4;
            const red = pixels[index];
            const green = pixels[index + 1];
            const blue = pixels[index + 2];
            const alpha = pixels[index + 3];
            const color = 'rgb(' + red + ',' + green + ',' + blue + ')';

            if(alpha > 0){
                this.particlesArray.push(new Particle(this, x, y, color)); |<<<----
            }
        }
    }
    draw(context){
        this.particlesArray.forEach(particle => particle.draw(context));
    }
    update(){
        this.particlesArray.forEach(particle => particle.update());
    }
}

```

`this` → representa a classe inteira que está sendo passada.

`x` → é a variável contadora da atualização do segundo loop, e ele representa a posição do eixo `x`.

`y` → é a variável contadora da atualização do primeiro loop, e ele representa a posição do eixo `y`.

`color` → é a variável que construímos no loop.

Agora é hora de começarmos a nossa animação.

16 TORNANDO IMAGENS EM UM SISTEMA DE PARTÍCULAS

Apenas para recapitular, nosso método `init()` vai rodar apenas uma vez quando a página carregar.

O que ele faz até o momento é:

1 - Desenha a imagem no `canvas`

2 - Depois usamos o `getImageData()` para analisar o `canvas` e conseguir pegar as informações dele. E selecionamos o array `data` devolvido.

3 - Colocamos ele em loops aninhados para converter esses valores e índices em coordenadas `x` e `y`.

4 - Utilizamos `width` e o `height` do `canvas` para saber até onde devemos fazer nossa análise.

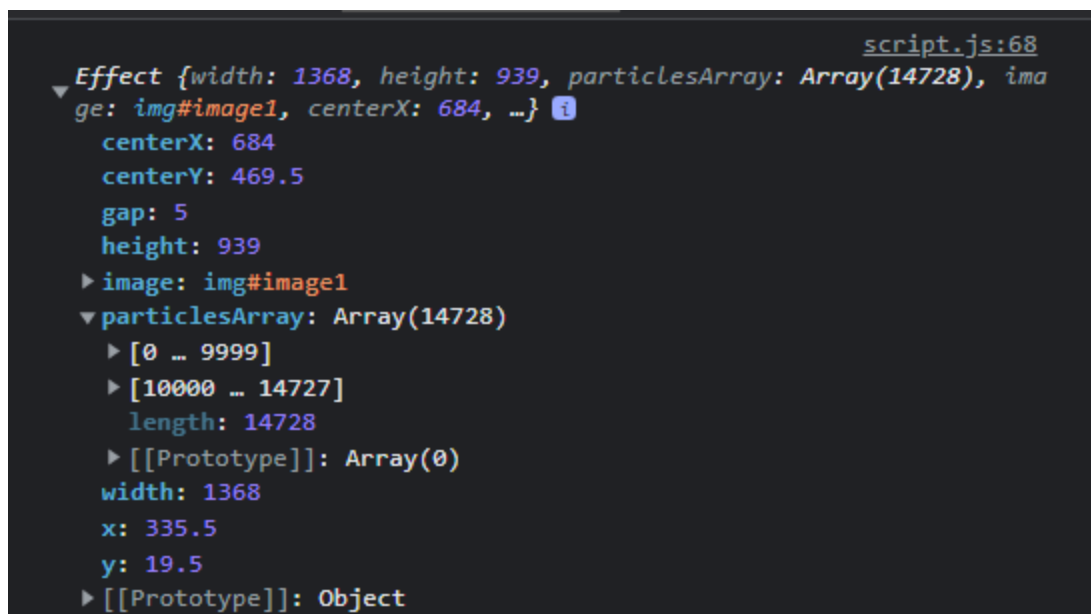
5 - Extraímos 4 valores: *Red*, *Green*, *Blue*, e *Alpha*.

6 - Concatenamos esses valores em uma notação RGB

6 - Testamos se a opacidade for maior que zero, e em caso positivo criamos uma partícula nova e passamos os parâmetros necessários sobre sua posição no canvas e suas cores.

Agora para testar se está tudo funcionando até agora, vamos imprimir a criação do nosso efeito:

```
const effect = new Effect(canvas.width, canvas.height);
effect.init(ctx);
console.log(effect);
```



Percebe-se que o nosso *array* possui 14728 partículas criadas a partir do nosso algoritmo de seleção produzido no capítulo anterior. Esse número pode variar de acordo com o tamanho da imagem que você utilizar ou de acordo com a resolução do *scan* que fizemos.

A resolução foi definida na propriedade `this.gap` e ela faz um *scan* de 5 x 5 pixels.

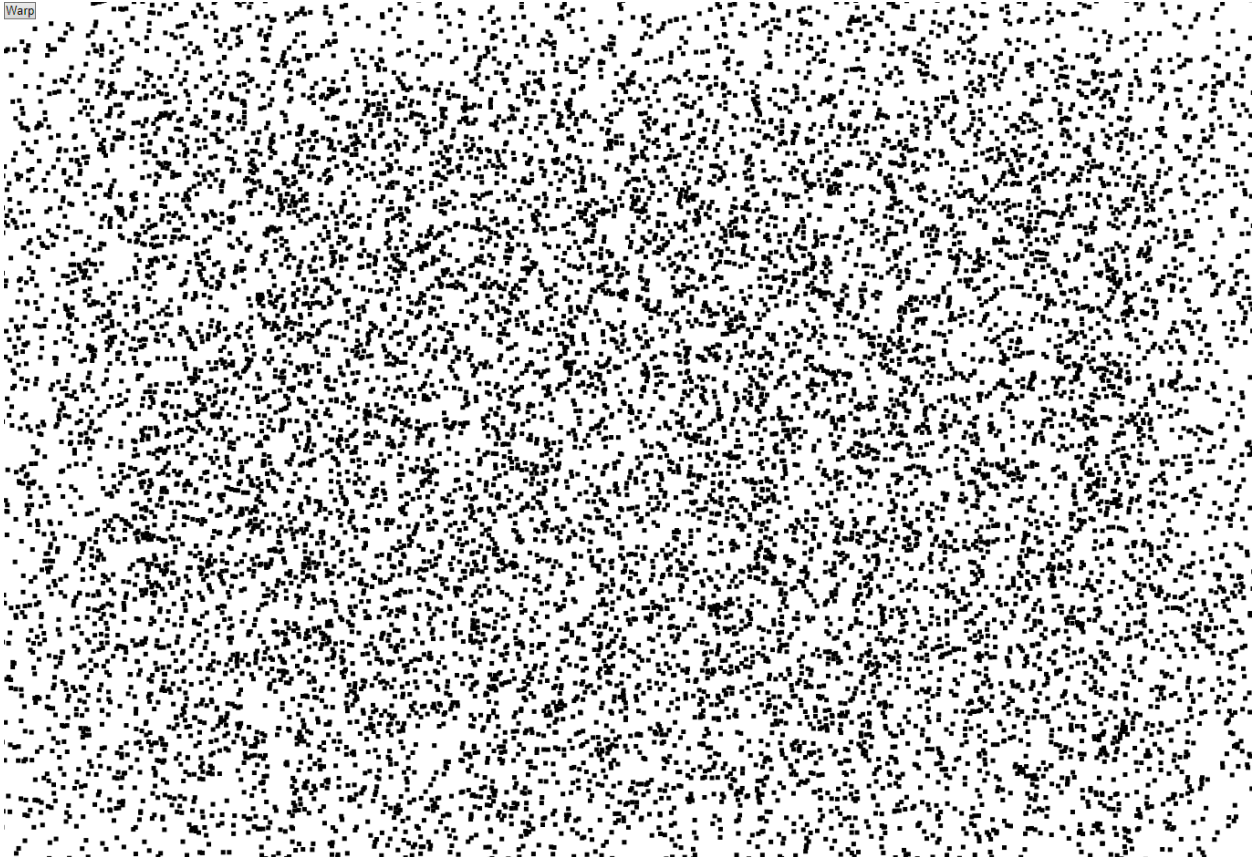
Se abrirmos qualquer objeto para análise, temos que eles possuem as propriedades que definimos no capítulo anterior (`originX`, `originY`, `Color`, `vx`, `vy`, `x` (posição x atual), `y` (posição y atual)):

```
▼ particlesArray: Array(14728)
  ► [0 ... 9999]
  ▼ [10000 ... 14727]
    ▼ [10000 ... 10099]
      ▼ 10000: Particle
        color: "rgb(197,107,127)"
        ► effect: Effect {width: 1368, height: 939, particlesArray
          originX: 770
          originY: 565
          size: 5
          vx: -0.7272968722222646
          vy: 0.6402939006345321
          x: 368.11045933572143
          y: 674.2461394627298
          ► [[Prototype]]: Object
```

Se algumas dessas propriedades estiverem sem nenhum valor, volte um pouco. Para o efeito funcionar precisamos que todas essas propriedades estejam com seus respectivos valores.

Vamos tirar o comentário da chamada da função `animate()` .

Warp



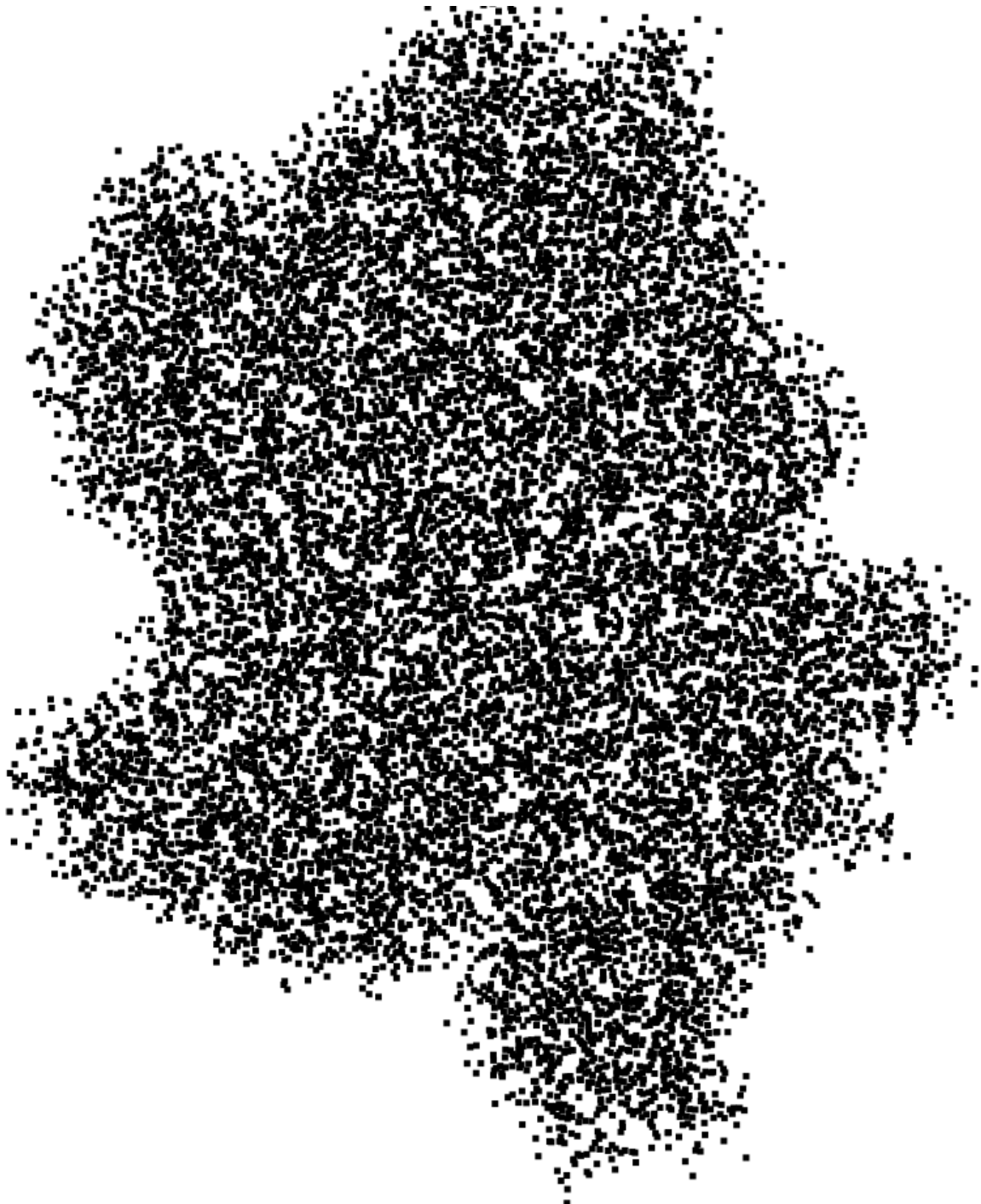
Se voltarmos na nossa página, veremos que nossa imagem não está mais lá e agora temos as nossas partículas flutuando por aí.

Vamos fazer nossas imagens se moverem para as suas posições de origem para recriar a imagem.

Para isso vamos alterar os valores da propriedade `this.x` e `this.y`, que antes eram aleatórios para `x` e `y`, que são os valores passados por parâmetro.

```
class Particle {
  constructor(effect, x, y, color){
    this.effect = effect;
    this.x = x; |<<<<----
    this.y = y; |<<<<----
    this.originX = Math.floor(x);
    this.originY = Math.floor(y);
    this.color = color;
    this.size = 5;
    this.vx = Math.random() * 2 - 1;
    this.vy = Math.random() * 2 - 1;
  }
  draw(context){
```

```
        context.fillRect(this.x, this.y, this.size, this.size);
    }
    update(){
        this.x += this.vx;
        this.y += this.vy;
    }
}
```



Agora precisamos fazer as partículas ficarem com sua respectiva cor.

Para conseguir fazer isso vamos utilizar a propriedade `fillStyle` do nosso `ctx`, e passar a cor que recebemos como parâmetro (repare que ele é uma propriedade e não uma função, portanto estamos reatribuindo seu valor):

```
class Particle {
  constructor(effect, x, y, color){
    this.effect = effect;
    this.x = x;
    this.y = y;
    this.originX = Math.floor(x);
    this.originY = Math.floor(y);
    this.color = color;
    this.size = 5;
    this.vx = Math.random() * 2 - 1;
    this.vy = Math.random() * 2 - 1;
  }
  draw(context){
    context.fillStyle = this.color; |<<<<----
    context.fillRect(this.x, this.y, this.size, this.size);
  }
  update(){
    this.x += this.vx;
    this.y += this.vy;
  }
}
```

E por último para evitar que as partículas fiquem se movendo vamos precisar alterar o `vx` e o `vy` para `0`:

```
class Particle {
  constructor(effect, x, y, color){
    this.effect = effect;
    this.x = x;
    this.y = y;
    this.originX = Math.floor(x);
    this.originY = Math.floor(y);
    this.color = color;
    this.size = 5;
    this.vx = 0; |<<<<----
    this.vy = 0; |<<<<----
  }
  draw(context){
    context.fillStyle(this.color);
    context.fillRect(this.x, this.y, this.size, this.size);
  }
  update(){
```



```
    this.x += this.vx;  
    this.y += this.vy;  
  }  
}
```

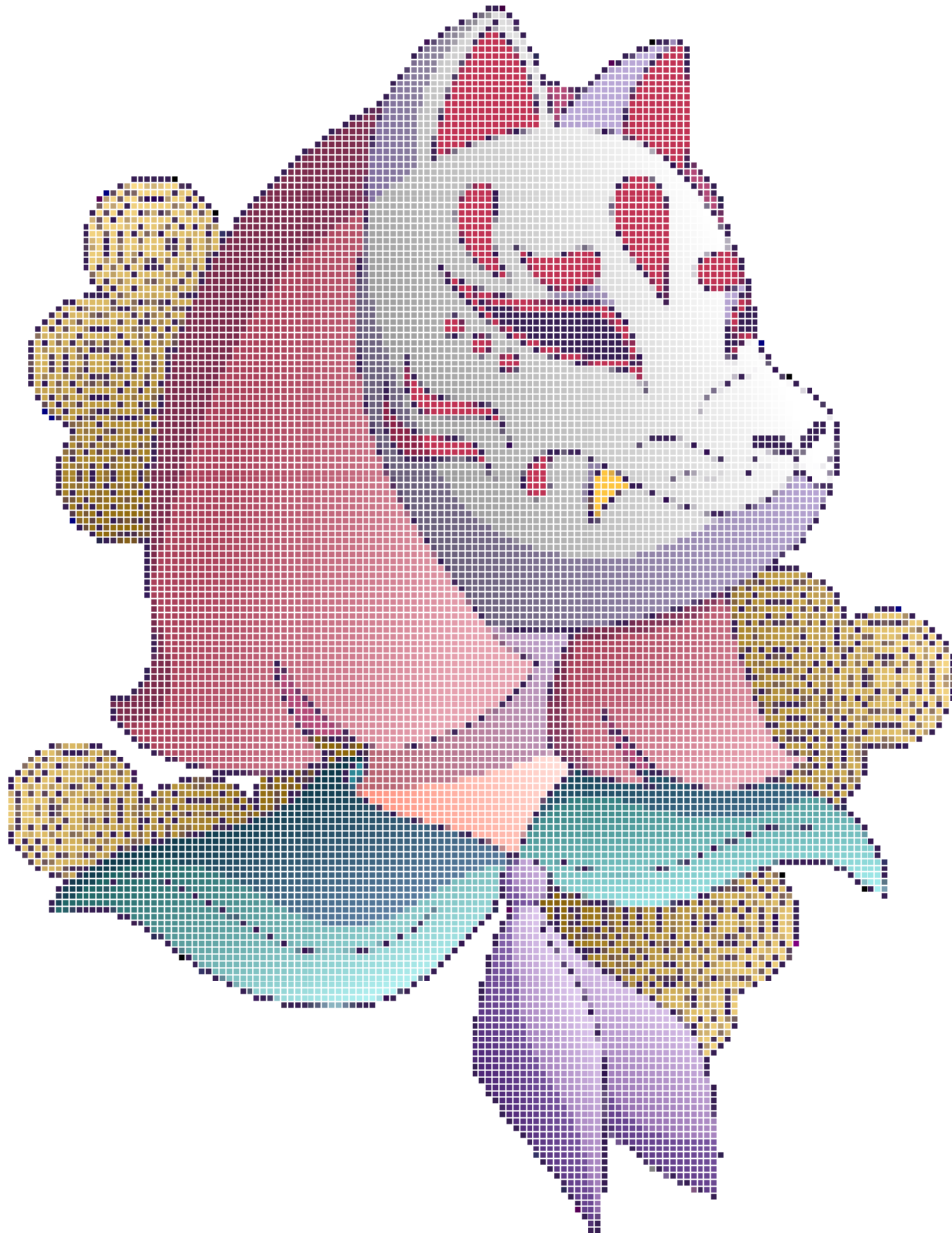
Assim:



E agora você sabe como recriar uma imagem no [canvas](#) como um sistema de partículas.

Se alterarmos a propriedade `this.size`, alteramos também a quantidade de detalhes que nossa imagem terá. Quanto menor seu valor mais processamento o efeito irá demandar.

Se alterarmos seu valor para 4 vamos ter um gap de 1 pixel entre as partículas.



Para evitar esse problema, basta que alteremos o valor do `this.size` para o mesmo valor do `this.gap` assim:

```

class Particle {
  constructor(effect, x, y, color){
    this.effect = effect;
    this.x = x;
    this.y = y;
    this.originX = Math.floor(x);
    this.originY = Math.floor(y);
    this.color = color;
    this.size = this.effect.gap; |<<<----
    this.vx = 0;
    this.vy = 0;
  }
  draw(context){
    context.fillStyle = this.color;
    context.fillRect(this.x, this.y, this.size, this.size);
  }
  update(){
    this.x += this.vx;
    this.y += this.vy;
  }
}

```

É interessante não colocar o valor do `this.gap` com números decimais por motivos de performance.

Também é melhor quando falamos de performance desenhar as partículas como retângulos ao invés de círculos. Em programas mais básicos você não sente tanto essa queda de performance, mas quando você tiver um sistema como esse, que possui mais de 3000 partículas por exemplo, você começa a sentir a diferença. Se quiser testar defina o valor do `this.gap` para 1 por exemplo. Por enquanto vamos definir o valor do `this.gap` para 3.

IMPORTANTE: se o seu computador começar a ter quedas de desempenho ou apresentar dificuldades para a renderização, você pode diminuir o tamanho da imagem ou aumentar o valor contido no `this.gap`.

17 TRANSIÇÃO ANIMADA DAS PARTÍCULAS

Queremos que as partículas sempre se movam para a sua posição inicial, assim quando nós as empurrarmos com o mouse ou quando quebrarmos a imagem em pedaços, as partículas vão sempre ser puxadas em direção ao seu ponto inicial.

Queremos que a imagem sempre se reconstrua quando que estiver quebrada. Para isso vamos escrever essa lógica dentro do método `update()` dentro da classe `Particle`.

Vamos começar de forma simples sem utilizar nenhuma física.

Para fazer isso, vamos precisar que as partículas saibam a diferença entre sua posição original e a sua posição atual.

```
update(){
  this.x += this.originX - this.x; |<<<<----
  this.y += this.originY - this.y; |<<<<----
}
```

Para lembrar, estamos chamando o método `update()` da classe `Effect` na função `animate()`:

```
function animate(){
  ctx.clearRect(0, 0, canvas.width, canvas.height);
  effect.draw(ctx);
  effect.update();
  requestAnimationFrame(animate);
}
```

Essa por sua vez chama a função `update()` da classe `Particle`, para cada elemento do *array*:

```
update(){
  this.particlesArray.forEach(particle => particle.update());
}
```

Agora vamos voltar nos valores que alteramos e vamos colocá-los entre parênteses, para que eles sejam diminuídos primeiro e depois somados:

```
update(){
  this.x += (this.originX - this.x); |<<<<----
  this.y += (this.originY - this.y); |<<<<----
}
```

Vamos agora colocar que a posição `x` e `y` é uma posição aleatória:

```
class Particle {
  constructor(effect, x, y, color){
```

```

    this.effect = effect;
    this.x = Math.random() * this.effect.width; |<<<----
    this.y = Math.random() * this.effect.height; |<<<----
    this.originX = Math.floor(x);
    this.originY = Math.floor(y);
    this.color = color;
    this.size = this.effect.gap;
    this.vx = 0;
    this.vy = 0;
  }
  draw(context){
    context.fillStyle = this.color;
    context.fillRect(this.x, this.y, this.size, this.size);
  }
  update(){
    this.x += (this.originX - this.x);
    this.y += (this.originY - this.y);
  }
}

```

Mas isso não causou nenhum efeito na página. Isso aconteceu, pois temos que alterar um pouquinho os valores no `update()` :

```

class Particle {
  constructor(effect, x, y, color){
    this.effect = effect;
    this.x = Math.random() * this.effect.width;
    this.y = Math.random() * this.effect.height;
    this.originX = Math.floor(x);
    this.originY = Math.floor(y);
    this.color = color;
    this.size = this.effect.gap;
    this.vx = 0;
    this.vy = 0;
  }
  draw(context){
    context.fillStyle = this.color;
    context.fillRect(this.x, this.y, this.size, this.size);
  }
  update(){
    this.x += (this.originX - this.x) * 0.1; |<<<----
    this.y += (this.originY - this.y) * 0.1; |<<<----
  }
}

```

Assim colocamos uma pequena diferença entre os valores, alterando a posição atual das partículas quando chamamos a animação na função `animate()`, e fazendo elas voltarem para o lugar de início.

Vamos agora aplicar um pouco de física.

Criaremos uma nova propriedade chamada `ease` (suavização), e substituiremos no lugar do `0.1`:

```
class Particle {
  constructor(effect, x, y, color){
    this.effect = effect;
    this.x = Math.random() * this.effect.width;
    this.y = Math.random() * this.effect.height;
    this.originX = Math.floor(x);
    this.originY = Math.floor(y);
    this.color = color;
    this.size = this.effect.gap;
    this.vx = 0;
    this.vy = 0;
    this.ease = 0.2; |<<<----
  }
  draw(context){
    context.fillStyle = this.color;
    context.fillRect(this.x, this.y, this.size, this.size);
  }
  update(){
    this.x += (this.originX - this.x) * this.ease; |<<<----
    this.y += (this.originY - this.y) * this.ease; |<<<----
  }
}
```

Alterando esse valor nós alteraremos o quão rápido, ou o quão devagar, nossa imagem vai se remontar.

Se você quiser algo em câmera lenta, pode diminuir ainda mais o valor para `0.01` ou qualquer outro. Se quiser uma velocidade aleatória também pode fazer utilizar o `Math.random()` entre várias outras coisas que podemos fazer.

Observe que para fazer toda essa lógica, não utilizamos nenhuma biblioteca externa e sim construímos do zero nós mesmos essa animação.

Podemos brincar com esse efeito de várias formas, alterando o valor `this.x` e `this.y`. Por exemplo se multiplicarmos o valor que tem lá dentro agora por algum valor maior que `1`, as partículas vão se espalhar muito mais, pois estamos definindo números maiores que o tamanho da tela. Se definirmos para `0` as partículas vão se concentrar em uma extremidade etc.

18 ANIMAÇÃO NO CLIQUE DO BOTÃO

Vamos criar um botão que quando for clicado vai randomizar as posições `x` e `y` e nós vamos ter uma animação bem legal das partículas remontando a imagem toda vez que clicarmos nele.

Vamos dar o nome de `warp()` na nossa classe `Particle`. Esse método vai alterar os valores das propriedades e podemos também reatribuir um valor para o `ease` só pra ver como fica:

```
class Particle {
  constructor(effect, x, y, color){
    this.effect = effect;
    this.x = Math.random() * this.effect.width;
    this.y = Math.random() * this.effect.height;
    this.originX = Math.floor(x);
    this.originY = Math.floor(y);
    this.color = color;
    this.size = this.effect.gap;
    this.vx = 0;
    this.vy = 0;
    this.ease = 0.2;
  }
  draw(context){
    context.fillStyle = this.color;
    context.fillRect(this.x, this.y, this.size, this.size);
  }
  update(){
    this.x += (this.originX - this.x) * this.ease;
    this.y += (this.originY - this.y) * this.ease;
  }
  warp(){
    this.x = Math.random() * this.effect.width; |<<<<----
    this.y = Math.random() * this.effect.height; |<<<<----
    this.ease = 0.05;                             |<<<<----
  }
}
```

Vamos agora na nossa classe `Effect` e vamos criar o mesmo método, a função desse vai ser passar por cada uma das partículas e aplicar os novos valores as propriedades de cada um deles. Ou seja, quando o botão for clicado ele vai chamar o método `warp()` da classe `Particle` para cada um dos elementos.

```
class Effect {
  constructor(width, height){
    this.width = width;
    this.height = height;
    this.particlesArray = [];
  }
}
```

```

        this.image = document.getElementById('image1');
        this.centerX = this.width * 0.5;
        this.centerY = this.height * 0.5;
        this.x = this.centerX - this.image.width * 0.5;
        this.y = this.centerY - this.image.height * 0.5;
        this.gap = 3;
    }
    init(context){
        context.drawImage(this.image, this.x, this.y);
        const pixels = context.getImageData(0, 0, this.width, this.height).data;
        for (let y = 0; y < this.height; y += this.gap){
            for(let x = 0; x < this.width; x += this.gap){
                const index = (y * this.width + x) * 4;
                const red = pixels[index];
                const green = pixels[index + 1];
                const blue = pixels[index + 2];
                const alpha = pixels[index + 3];
                const color = 'rgb(' + red + ',' + green + ',' + blue + ')';

                if(alpha > 0){
                    this.particlesArray.push(new Particle(this, x, y, color));
                }
            }
        }
    }
    draw(context){
        this.particlesArray.forEach(particle => particle.draw(context));
    }
    update(){
        this.particlesArray.forEach(particle => particle.update());
    }
    warp(){
        this.particlesArray.forEach(particle => particle.warp());
    }
}

```

Agora precisamos selecionar o botão que criamos lá no início desse tutorial, e atribuir um evento para ele, que chamará esse último método `warp()` da classe `Effect` que criamos.

```

const warpButton = document.getElementById('warpButton');
warpButton.addEventListener('click', function(){
    effect.warp();
})

```

Lembre-se que já criamos o objeto `effect` anteriormente por isso podemos ter acesso aos seus métodos quando o referenciamos novamente. Pronto, agora já está tudo

funcionando.

Vamos apenas alterar o estilo do botão para que fique mais bonito:

```
button {
  padding: 10px;
  font-size: 40px;
  margin: 20px;
  font-family: 'Bangers', cursive;
  color: white;
  background-color: black;
  transition: 0.2s;
}

button:hover{
  color: black;
  background-color: white;
  cursor: pointer;
}
```

Também adicionamos uma fonte que deve ser colocada no `head` do HTML, antes do link do CSS para que a fonte esteja disponível no arquivo CSS, dessa forma:

```
<head>
  ...
  <link rel="preconnect" href="https://fonts.googleapis.com">
  <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
  <link href="https://fonts.googleapis.com/css2?family=Bangers&display=swap" rel="stylesheet">
  <link rel="stylesheet" href="style.css">
  ...
</head>
```

Aqui você pode colocar a fonte que mais te agrada, aliás a estilização toda pode ser alterada, isso não fará diferença no resultado final.

19 INTERAÇÕES COM O MOUSE E A FÍSICA DAS PARTÍCULAS

Para conseguir fazer com que as partículas se movam de acordo com o mouse precisamos que a classe `Effect` saiba aonde o mouse está.

Para isso, vamos criar uma propriedade `this.mouse` que será um objeto e ele terá 3 propriedades:

`radius` → define a área ao redor do cursor do mouse, aonde as partículas vão começar a reagir com ele. Vamos definir seu valor como `3000`. Não, esse valor não significa `3000` pixels, vamos explicar mais tarde o motivo desse valor ser tão alto, mas em resumo é por motivos de performance.

`x` e `y` → vão começar com os valores `undefined`.

```
class Effect {
  constructor(width, height){
    this.width = width;
    this.height = height;
    this.particlesArray = [];
    this.image = document.getElementById('image1');
    this.centerX = this.width * 0.5;
    this.centerY = this.height * 0.5;
    this.x = this.centerX - this.image.width * 0.5;
    this.y = this.centerY - this.image.height * 0.5;
    this.gap = 3;
    this.mouse = {      |<<<<----
      radius: 3000, |<<<<----
      x: undefined, |<<<<----
      y: undefined, |<<<<----
    }
  }
}
```

O `constructor()` da classe é executado quando a classe é instanciada (um novo objeto é criado) pela palavra chave `new`, podemos tomar vantagem sobre isso e podemos adicionar qualquer trecho de código dentro desse método que queremos que seja executado, no momento em que a classe `Effect` é instanciada.

Podemos até mesmo adicionar um escutador de eventos dentro dele, e é o que nós vamos fazer.

Vamos monitorar por esse escutador o evento `mousemove` e vamos ligar ele ao seu escopo léxico (É o contexto atual de execução, em que valores e expressões são "visíveis" ou podem ser referenciadas. Se uma variável ou outra expressão não estiver "no **escopo** atual", então não está disponível para uso).

Assim, tendo certeza dessa ligação poderemos alterar os valores de `x` e `y` quando o evento estiver ativo.

```
class Effect {
  constructor(width, height){
```

```

    this.width = width;
    this.height = height;
    this.particlesArray = [];
    this.image = document.getElementById('image1');
    this.centerX = this.width * 0.5;
    this.centerY = this.height * 0.5;
    this.x = this.centerX - this.image.width * 0.5;
    this.y = this.centerY - this.image.height * 0.5;
    this.gap = 3;
    this.mouse = {
        radius: 3000,
        x: undefined,
        y: undefined,
    }
    window.addEventListener('mousemove', function(){ |<<<<----

    })
}
...
}

```

Para ter certeza de que a função que será disparada quando o evento acontecer e se lembrar aonde dentro do objeto ela foi definida, podemos utilizar um método especial do Java Script de ligação ou podemos mudar a função que será chamada para uma função `es6 arrow function`, que também pode ser chamada de `fat arrow function`.

```

class Effect {
    constructor(width, height){
        this.width = width;
        this.height = height;
        this.particlesArray = [];
        this.image = document.getElementById('image1');
        this.centerX = this.width * 0.5;
        this.centerY = this.height * 0.5;
        this.x = this.centerX - this.image.width * 0.5;
        this.y = this.centerY - this.image.height * 0.5;
        this.gap = 3;
        this.mouse = {
            radius: 3000,
            x: undefined,
            y: undefined,
        }
        window.addEventListener('mousemove', () => { |<<<<----

        })
    }
    ...
}

```

Uma das vantagens de se utilizar uma `arrow function` é que elas automaticamente ligam essa palavra chave ao contexto léxico do código que está ao seu redor. Então ela sempre vai conseguir ver a propriedade `this.mouse` que é exatamente o que precisamos aqui.

Para ser mais exato, `arrow functions` herda o `this` do seu escopo léxico “pai”. Dentro de uma `arrow function` a palavra `this` sempre representa o objeto no qual ela foi definida.

Passado essas explicações, vamos passar o evento como um parâmetro, e como só vamos ter esse parâmetro nessa função, podemos até tirar os parênteses:

```
class Effect {
  constructor(width, height){
    this.width = width;
    this.height = height;
    this.particlesArray = [];
    this.image = document.getElementById('image1');
    this.centerX = this.width * 0.5;
    this.centerY = this.height * 0.5;
    this.x = this.centerX - this.image.width * 0.5;
    this.y = this.centerY - this.image.height * 0.5;
    this.gap = 3;
    this.mouse = {
      radius: 3000,
      x: undefined,
      y: undefined,
    }
    window.addEventListener('mousemove', event => { |<<<----
  })
}
...
}
```

NOTA: se você quiser saber o que é esse parâmetro, pode dar um `console.log` nele logo abaixo. Mas em resumo ele é um objeto que é passado pelo “escutador” de eventos toda vez que o evento é disparado. Nele existem propriedades muito importantes e uteis, como qual botão foi apertado, se o mouse apertou com o botão 1 ou 2, e o que vamos utilizar aqui, a coordenada `x` e `y` de onde o mouse se encontra. Vamos alterar o valor de `x` e `y` para onde o valor do mouse atualmente estiver:

```

class Effect {
  constructor(width, height){
    this.width = width;
    this.height = height;
    this.particlesArray = [];
    this.image = document.getElementById('image1');
    this.centerX = this.width * 0.5;
    this.centerY = this.height * 0.5;
    this.x = this.centerX - this.image.width * 0.5;
    this.y = this.centerY - this.image.height * 0.5;
    this.gap = 3;
    this.mouse = {
      radius: 3000,
      x: undefined,
      y: undefined,
    }
    window.addEventListener('mousemove', event => {
      this.mouse.x = event.x;           |<<<<----
      this.mouse.y = event.y;           |<<<<----
      console.log(this.mouse.x, this. mouse.y); |<<<<----
    })
  }
  ...
}

```

Se movimentarmos o mouse teremos:

389	372	script.js:54
389	371	script.js:54
300	380	script.js:54
236	418	script.js:54
203	450	script.js:54
215	488	script.js:54
274	527	script.js:54
275	527	script.js:54
384	560	script.js:54
457	568	script.js:54
559	559	script.js:54
693	502	script.js:54
695	502	script.js:54
797	423	script.js:54
882	357	script.js:54
944	327	script.js:54
945	327	script.js:54
1005	323	script.js:54
1006	323	script.js:54
1058	328	script.js:54
1059	328	script.js:54
1087	347	script.js:54
1106	373	script.js:54
1123	396	script.js:54
1129	413	script.js:54
1132	426	script.js:54
1133	428	script.js:54
1135	432	script.js:54
1140	434	script.js:54

Assim, as propriedades `x` e `y` da posição atual do mouse estão sendo alterados, e ainda como estamos imprimindo as propriedades da classe (`this.mouse.x` e `this.mouse.y`) e não do evento sabemos que elas estão sendo gravadas corretamente.

Podemos deletar esse `console.log` agora. Tanto o que imprime o objeto `effect` quanto esse que colocamos para testar se as coordenadas estavam sendo corretamente gravadas.

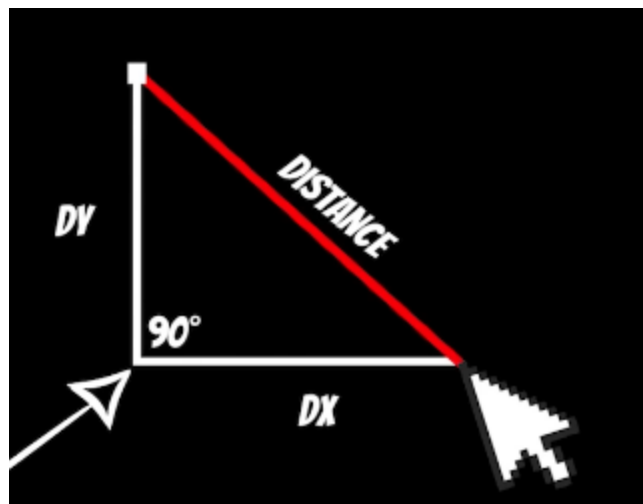
Queremos que cada partícula agora, fique checando o quão distantes elas estão do mouse, e se o mouse estiver próximo o suficiente queremos que a partícula seja empurrada para longe dele.

Vamos escrever essa lógica dentro do método `update()` que fica na classe `Particle`.

Vamos criar 2 propriedades novas `dx` e `dy` que serão a distância no eixo `x` da partícula para o mouse, e a mesma coisa no eixo `y`.

```
class Particle {  
  ...  
  update(){  
    this.dx = this.effect.mouse.x - this.x; |<<<----  
    this.dy = this.effect.mouse.y - this.y; |<<<----  
    this.x += (this.originX - this.x) * this.ease;  
    this.y += (this.originY - this.y) * this.ease;  
  }  
  ...  
}
```

Para calcular a distância entre dois pontos podemos utilizar o teorema de Pitágoras.



Criamos o `dy` e o `dx` e eles servirão para nos ajudar a achar a distância.

A distância será igual a hipotenusa.

Para achar a hipotenusa podemos utilizar: $C = \sqrt{A^2 + B^2}$

convertido para o Java Script fica:

```

class Particle {
  ...
  update(){
    this.dx = this.effect.mouse.x - this.x;
    this.dy = this.effect.mouse.y - this.y;
    this.distance = Math.sqrt(this.dx * this.dx + this.dy * this.dy); |<<<<----

    this.x += (this.originX - this.x) * this.ease;
    this.y += (this.originY - this.y) * this.ease;
  }
  ...
}

```

Porém nós não vamos utilizar o `Math.sqrt` (basicamente devolve a raiz quadrada) pois ele é muito pesado em questão de performance. Vamos apenas precisar alterar o valor do `this.mouse.radius` para valores maiores, para que não seja necessário utilizarmos esse *square root*.

```

class Particle {
  ...
  update(){
    this.dx = this.effect.mouse.x - this.x;
    this.dy = this.effect.mouse.y - this.y;
    this.distance = this.dx * this.dx + this.dy * this.dy; |<<<<----

    this.x += (this.originX - this.x) * this.ease;
    this.y += (this.originY - this.y) * this.ease;
  }
  ...
}

```

E foi por isso que colocamos o valor para `3000` no `this.mouse.radius`. Você vai poder ver o tamanho do range quando fizermos a animação funcionar em alguns instantes.

Vamos implementar um pouco de física: as partículas que estiverem mais próximas do mouse serão empurradas com mais força. Criaremos uma propriedade chamada `this.force` ele será o raio do mouse, ou seja, a área ao redor do mouse onde as partículas reagirão quando estiverem próximas, dividido entre a distância entre o mouse e a partícula.

```

class Particle {
  ...
  update(){

```



```

        this.dx = this.effect.mouse.x - this.x;
        this.dy = this.effect.mouse.y - this.y;
        this.distance = this.dx * this.dx + this.dy * this.dy;
        this.force = -this.effect.mouse.radius / this.distance; |<<<----

        this.x += (this.originX - this.x) * this.ease;
        this.y += (this.originY - this.y) * this.ease;
    }
    ...
}

```

Para que as partículas sejam empurradas na direção correta precisamos colocar um menos na frente do primeiro valor. Quando conseguirmos fazer a animação teste trocar esse `-` por um `+` para ver o que acontece.

Vamos checar se a distância é menor do que o raio do mouse, em caso positivo, vamos empurrar a partícula para longe do mouse.

Mas antes de fazermos isso, estamos utilizando propriedades que não foram definidas no `constructor()` então vamos fazer isso:

```

class Particle {
    constructor(effect, x, y, color){
        this.effect = effect;
        this.x = Math.random() * this.effect.width;
        this.y = Math.random() * this.effect.height;
        this.originX = Math.floor(x);
        this.originY = Math.floor(y);
        this.color = color;
        this.size = this.effect.gap;
        this.vx = 0;
        this.vy = 0;
        this.ease = 0.2;
        this.dx = 0; |<<<----
        this.dy = 0; |<<<----
        this.distance = 0; |<<<----
        this.force = 0; |<<<----
        this.angle = 0; |<<<----
    }
    ...
    update(){
        this.dx = this.effect.mouse.x - this.x;
        this.dy = this.effect.mouse.y - this.y;
        this.distance = Math.sqrt(this.dx * this.dx + this.dy * this.dy);
        this.force = -this.effect.mouse.radius / this.distance;

        if(this.distance < this.effect.mouse.radius){ |<<<----

    }
}

```

```

        this.x += (this.originX - this.x) * this.ease;
        this.y += (this.originY - this.y) * this.ease;
    }
    ...
}

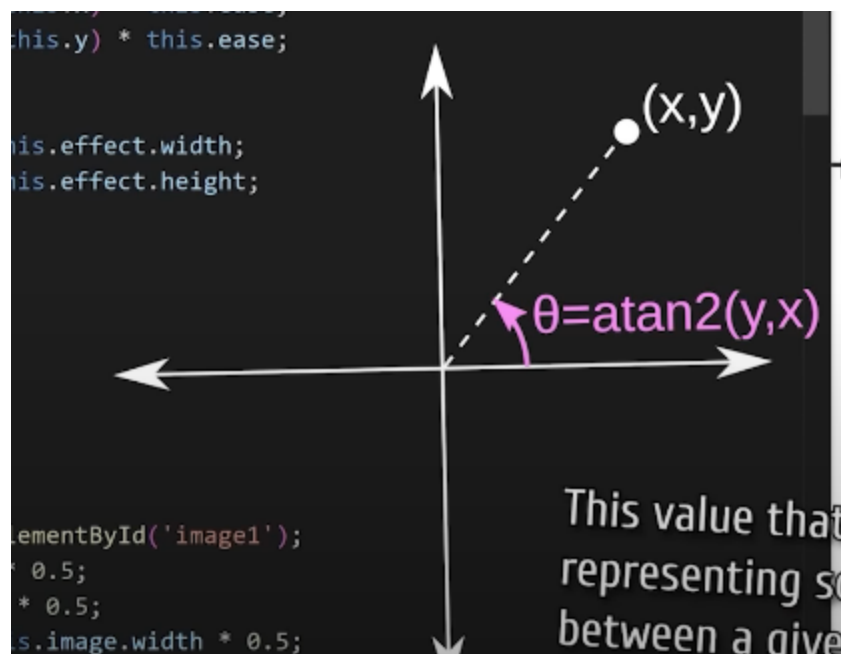
```

Criamos uma nova propriedade também chamada `this.angle`. Essa propriedade vai determinar qual a direção que as partículas serão empurradas, quando elas interagirem com o mouse.

Então nós vamos calcular esse ângulo dentro da nossa estrutura condicional que testa se a distância entre o mouse e a partícula é menor do que o raio do mouse.

Vamos utilizar um outro método do `Math` que é o `math.atan2()`. Esse método retorna um valor numérico em radianos entre $-\pi$ e $+\pi$. Esse valor que é retornado também é chamado de ângulo theta θ .

Ou seja, ele vai retornar um valor em radianos da distância entre um ponto e o eixo x positivo:



Esse método espera primeiro o ponto `y` e depois o `x`.

```

class Particle {
    constructor(effect, x, y, color){

```

```

    this.effect = effect;
    this.x = Math.random() * this.effect.width;
    this.y = Math.random() * this.effect.height;
    this.originX = Math.floor(x);
    this.originY = Math.floor(y);
    this.color = color;
    this.size = this.effect.gap;
    this.vx = 0;
    this.vy = 0;
    this.ease = 0.2;
    this.dx = 0;
    this.dy = 0;
    this.distance = 0;
    this.force = 0;
    this.angle = 0;
  }
  ...
  update(){
    this.dx = this.effect.mouse.x - this.x;
    this.dy = this.effect.mouse.y - this.y;
    this.distance = Math.sqrt(this.dx * this.dx + this.dy * this.dy);
    this.force = -this.effect.mouse.radius / this.distance;

    if(this.distance < this.effect.mouse.radius){
      this.angle = Math.atan2(this.dy, this.dx); |<<<----
    }

    this.x += (this.originX - this.x) * this.ease;
    this.y += (this.originY - this.y) * this.ease;
  }
  ...
}

```

Então a diferença entre as coordenadas `x` e `y` da partícula e as coordenadas `x` e `y` do cursor do mouse, representadas aqui pelo `dy` e o `dx`, nos retornam o valor de um ângulo.

Também vamos aumentar a velocidade `vx` e `vy` de acordo com a `this.force`, esta que vai depender do raio entre o raio do mouse e a distância entre o mouse e a partícula, fazendo assim as partículas que estiver mais próximas serem empurradas com mais velocidade do que as que estiverem mais longe.

E para saber qual ângulo as partículas serão empurradas vamos passar o ângulo que calculamos antes como parâmetro para o método `Math.cos()` que vai mapear o ângulo que passamos em radianos em um valor entre -1 e +1. Esse valor vai representar o cosseno do ângulo e vai fazer as partículas flutuarem ao longo dos limites do círculo em volta do mouse.

```

class Particle {
  constructor(effect, x, y, color){
    this.effect = effect;
    this.x = Math.random() * this.effect.width;
    this.y = Math.random() * this.effect.height;
    this.originX = Math.floor(x);
    this.originY = Math.floor(y);
    this.color = color;
    this.size = this.effect.gap;
    this.vx = 0;
    this.vy = 0;
    this.ease = 0.2;
    this.dx = 0;
    this.dy = 0;
    this.distance = 0;
    this.force = 0;
    this.angle = 0;
  }
  ...
  update(){
    this.dx = this.effect.mouse.x - this.x;
    this.dy = this.effect.mouse.y - this.y;
    this.distance = Math.sqrt(this.dx * this.dx + this.dy * this.dy);
    this.force = -this.effect.mouse.radius / this.distance;

    if(this.distance < this.effect.mouse.radius){
      this.angle = Math.atan2(this.dy, this.dx);
      this.vx = this.force * Math.cos(this.angle); |<<<<----
    }

    this.x += (this.originX - this.x) * this.ease;
    this.y += (this.originY - this.y) * this.ease;
  }
  ...
}

```

`Math.cos()` e `Math.atan2()` podem ser um pouco difíceis de entender se você estiver vendo pela primeira vez. Não é tão importante assim entender eles, desde que você entenda como utilizá-los, e conseguir a animação que você quer. Quando estivermos com a animação funcionando, vamos brincar com os valores para ver o que eles fazem.

O feedback visual da animação pode nos ajudar a entender melhor. O que fizemos agora é difícil, então não se sinta triste se não conseguiu acompanhar toda a lógica trigonométrica que aplicamos aqui, isso não vai afetar sua habilidade em ser um bom desenvolvedor em Java Script.

Assim, nós atualizamos os valores de velocidade e de direção que as partículas serão empurradas no eixo x. Vamos fazer a mesma coisa para `vy` porém agora vamos utilizar o `Math.sin()`. Passando o mesmo ângulo, o `Math.sin()` e o `Math.cos()`, vão trabalhar em conjunto para mapear a posição ao longo do raio de um círculo.

Agora podemos utilizar o `vx` e o `vy` nas posições `x` e `y` dessa forma:

```
class Particle {
  ...
  update(){
    this.dx = this.effect.mouse.x - this.x;
    this.dy = this.effect.mouse.y - this.y;
    this.distance = Math.sqrt(this.dx * this.dx + this.dy * this.dy);
    this.force = -this.effect.mouse.radius / this.distance;

    if(this.distance < this.effect.mouse.radius){
      this.angle = Math.atan2(this.dy, this.dx);
      this.vx = this.force * Math.cos(this.angle);
      this.vy = this.force * Math.sin(this.angle);      |<<<<----
    }

    this.x += this.vx + (this.originX - this.x) * this.ease; |<<<<----
    this.y += this.vy + (this.originY - this.y) * this.ease; |<<<<----
  }
  ...
}
```

Teste um pouco e veja o que está acontecendo.

Para melhorar um pouco nossa física, também podemos adicionar um pouco de atrito, para isso vamos adicionar uma nova propriedade no `constructor()` da classe `Particle`, depois vamos multiplicar pelas velocidades:

```
class Particle {
  constructor(effect, x, y, color){
    this.effect = effect;
    this.x = Math.random() * this.effect.width;
    this.y = Math.random() * this.effect.height;
    this.originX = Math.floor(x);
    this.originY = Math.floor(y);
    this.color = color;
    this.size = this.effect.gap;
    this.vx = 0;
    this.vy = 0;
    this.ease = 0.2;
    this.friction = 0.95; |<<<<----
    this.dx = 0;
```

```

        this.dy = 0;
        this.distance = 0;
        this.force = 0;
        this.angle = 0;
    }
    ...
    update(){
        this.dx = this.effect.mouse.x - this.x;
        this.dy = this.effect.mouse.y - this.y;
        this.distance = Math.sqrt(this.dx * this.dx + this.dy * this.dy);
        this.force = -this.effect.mouse.radius / this.distance;

        if(this.distance < this.effect.mouse.radius){
            this.angle = Math.atan2(this.dy, this.dx);
            this.vx = this.force * Math.cos(this.angle);
            this.vy = this.force * Math.sin(this.angle);
        }

        this.x += (this.vx * this.friction) + (this.originX - this.x) * this.ease; |<--
        this.y += (this.vy * this.friction) + (this.originY - this.y) * this.ease; |<--
    }
    ...
}

```

Por fim algumas partículas estão se perdendo assim, faça essa correção que elas pararão.

```

class Particle {
    constructor(effect, x, y, color){
        this.effect = effect;
        this.x = Math.random() * this.effect.width;
        this.y = Math.random() * this.effect.height;
        this.originX = Math.floor(x);
        this.originY = Math.floor(y);
        this.color = color;
        this.size = this.effect.gap;
        this.vx = 0;
        this.vy = 0;
        this.ease = 0.2;
        this.friction = 0.95;
        this.dx = 0;
        this.dy = 0;
        this.distance = 0;
        this.force = 0;
        this.angle = 0;
    }
    ...
    update(){
        this.dx = this.effect.mouse.x - this.x;
        this.dy = this.effect.mouse.y - this.y;
    }
}

```

```

    this.distance = Math.sqrt(this.dx * this.dx + this.dy * this.dy);
    this.force = -this.effect.mouse.radius / this.distance;

    if(this.distance < this.effect.mouse.radius){
        this.angle = Math.atan2(this.dy, this.dx);
        this.vx = this.force * Math.cos(this.angle);
        this.vy = this.force * Math.sin(this.angle);
    }

    this.x += (this.vx *= this.friction) + (this.originX - this.x) * this.ease; |<<-
    this.y += (this.vy *= this.friction) + (this.originY - this.y) * this.ease; |<<-
}
...
}

```

Brinque com os valores e quebre os códigos para ter diferentes animações.