Prática de construção de objetos

Começando

Este exemplo fará uso da Canvas API, para desenhar as bolas na tela, e da requestAnimationFrame API para animar toda a exibição

Para começar, faça cópias locais de nossos arquivos index.html, style.css, e main.js. Estes contêm o seguinte, respectivamente:

- Um documento HTML muito simples com um elemento `<h1>` , um elemento `<canvas>` para desenhar nossas bolas e elementos para aplicar nosso CSS e JavaScript em nosso HTML.
- Alguns estilos muito simples, que servem principalmente para estilizar e posicionar o `<h1>`, e se livrar de qualquer barra de rolagem ou margem ao redor da borda da página (para que fique bonito e arrumado).
- Um JavaScript que serve para configurar o elemento `<canvas>` e fornecer uma função geral que vamos usar.

A primeira parte do script é assim:

Canva

```
const canvas = document.querySelector('canvas');

const ctx = canvas.getContext('2d');

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

Modelando uma bola no nosso programa

Nosso programa contará com muitas bolas saltando ao redor da tela. Como todas essas bolas se comportarão da mesma maneira, faz sentido representá-las com um objeto. Vamos começar adicionando o construtor a seguir ao final do código.

Ball

```
function Ball(x, y, velX, velY, color, size) {
  this.x = x;
  this.y = y;
  this.velX = velX;
  this.velY = velY;
  this.color = color;
  this.size = size;
}
```

Isso classifica as propriedades, mas e os métodos? Queremos realmente fazer com que nossas bolas façam algo em nosso programa.

Desenhando a bola

Primeiro adicione o seguinte método `draw()` ao `prototype` do `Ball()` :

draw

```
Ball.prototype.draw = function() {
  ctx.beginPath();
  ctx.fillStyle = this.color;
  ctx.arc(this.x, this.y, this.size, 0, 2 * Math.PI);
  ctx.fill();
}
```

Usando esta função, podemos dizer a nossa bola para desenhar-se na tela, chamando uma série de membros do contexto de tela 2D que definimos anteriormente (ctx). O contexto é como o papel, e agora queremos comandar nossa caneta para desenhar algo nela:

- Primeiro, usamos beginPath() para declarar que queremos desenhar uma forma no papel.
- Em seguida, usamos fillStyle para definir a cor que queremos que a forma seja — nós a definimos como a propriedade color da nossa bola.
- Em seguida, usamos o método arc() para traçar uma forma de arco no papel. Seus parâmetros são:
- A posição x e y do centro do arco — estamos especificando as propriedades x e y da nossa bola.
- O raio do nosso arco — estamos especificando a propriedade size da nossa bola.
- Os dois últimos parâmetros especificam o número inicial e final de graus em volta do círculo em que o arco é desenhado entre eles. Aqui nós especificamos 0 graus e 2 * PI, que é o equivalente a 360 graus em radianos (irritantemente, você tem que especificar isso em radianos). Isso nos dá um círculo completo. Se você tivesse especificado apenas 1 * PI, você obteria um semicírculo (180 graus).

- Por último, usamos o método fill(), que basicamente diz "terminar de desenhar o caminho que começamos com `beginPath()`, e preencher a área que ocupa com a cor que especificamos anteriormente em `fillStyle` ."

Vamos testar

teste

```
let testBall = new Ball(50, 100, 4, 4, 'blue', 10);

testBall.x
testBall.size
testBall.color
testBall.draw()
```

Atualizando os dados da bola

Podemos desenhar a bola na posição, mas para começar a mover a bola, precisamos de uma função de atualização de algum tipo. Adicione o seguinte código na parte inferior do seu arquivo JavaScript, para adicionar um método `update()` ao `prototype` do `Ball()` :

update

```
Ball.prototype.update = function() {
  if ((this.x + this.size) >= width) {
    this.velX = -(this.velX);
  }

  if ((this.x - this.size) <= 0) {
    this.velX = -(this.velX);
  }

  if ((this.y + this.size) >= height) {
    this.velY = -(this.velY);
  }

  if ((this.y - this.size) <= 0) {
    this.velY = -(this.velY);
  }

  this.x += this.velX;
  this.y += this.velY;
}
```

As primeiras quatro partes da função verificam se a bola atingiu a borda da tela. Se tiver, invertemos a polaridade da velocidade relevante para fazer a bola viajar na direção oposta.

Assim, por exemplo, se a bola estava viajando para cima (positivo `title`), então a velocidade vertical é alterada de forma que ela comece a viajar para baixo (negativo `velY`).

Animando a bola

Vamos começar a adicionar bolas à tela e a animá-las.

Primeiro, precisamos de criar um lugar para armazenar todas as nossas bolas e então preenche-lo. O código a seguir fará esse trabalho — adicione-o ao final do seu código agora:

balls

```
let balls = [];

while (balls.length < 25) {
  let size = random(10,20);
  let ball = new Ball(
    // ball position always drawn at least one ball width
    // away from the edge of the canvas, to avoid drawing errors
    random(0 + size,width - size),
    random(0 + size,height - size),
    random(-7,7),
    random(-7,7),
    'rgb(' + random(0,255) + ',' + random(0,255) + ',' + random(0,255)
+')',
    size
  );

  balls.push(ball);
}
```

O laço while cria uma nova instância da nossa `Ball()` usando valores aleatórios gerados com nossa função `random()`, então a função `push()` coloca ela no final do nosso array de bolas, mas somente enquanto o número de bolas no array é menor que 25.

Adicione o seguinte ao final do seu código agora:

loop

```
function loop() {
  ctx.fillStyle = 'rgba(0, 0, 0, 0.25)';
  ctx.fillRect(0, 0, width, height);

  for (let i = 0; i < balls.length; i++) {
    balls[i].draw();
    balls[i].update();
  }
```

```
  requestAnimationFrame(loop);
}
```

Todos os programas que animam as coisas geralmente envolvem um loop de animação, que serve para atualizar as informações no programa e renderizar a visualização resultante em cada quadro da animação; esta é a base para a maioria dos jogos e outros programas. Nossa função `title` faz o seguinte:

- Define a cor de preenchimento da tela como preto semitransparente e desenha um retângulo com a cor em toda a largura e altura da tela, usando `fillRect()` (os quatro parâmetros fornecem uma coordenada de início e uma largura e altura para o retângulo desenhado). Isso serve para encobrir o desenho do quadro anterior antes que o próximo seja desenhado. Se você não fizer isso, você verá apenas longas cobras se movimentando ao redor da tela, em vez de mover as bolas! A cor do preenchimento é definida como semitransparente, rgba(0,0,0,0.25), para permitir que os poucos quadros anteriores brilhem levemente, produzindo as pequenas trilhas atrás das bolas à medida que elas se movem. Se você mudou 0,25 para 1, você não vai mais vê-los. Tente variar esse número para ver o efeito que ele tem.
- Percorre todas as bolas no array `balls` e executa a função `draw()` e `update()` de cada bola para desenhar cada uma delas na tela, depois faz as atualizações necessárias para a posição e a velocidade a tempo para o próximo quadro.
- Executa a função novamente usando o método `requestAnimationFrame()` — quando esse método é executado constantemente e passa o mesmo nome de função, ele executará essa função um número definido de vezes por segundo para criar uma animação suave. Isso geralmente é feito de forma recursiva — o que significa que a função está chamando a si mesma toda vez que é executada, portanto, ela será executada repetidas vezes.

Por último, adicione a seguinte linha à parte inferior do seu código — precisamos chamar a função uma vez para iniciar a animação.

loop

```
loop();
```

Adicionando detecção de colisão

Agora, para um pouco de diversão, vamos adicionar alguma detecção de colisão ao nosso programa, para que nossas bolas saibam quando bateram em outra bola.

Primeiro de tudo, adicione a seguinte definição de método abaixo onde você definiu o método `update()` (ou seja, o bloco `Ball.prototype.update`).

collision

```
Ball.prototype.collisionDetect = function() {
  for (let j = 0; j < balls.length; j++) {
    if (!(this === balls[j])) {
      const dx = this.x - balls[j].x;
      const dy = this.y - balls[j].y;
      const distance = Math.sqrt(dx * dx + dy * dy);

      if (distance < this.size + balls[j].size) {
        balls[j].color = this.color = 'rgb(' + random(0, 255) + ',' +
random(0, 255) + ',' + random(0, 255) +')';
      }
    }
  }
}
```

Esse método é um pouco complexo, então não se preocupe se você não entender exatamente como isso funciona agora. Uma explicação a seguir:

- Para cada bola, precisamos checar todas as outras bolas para ver se ela colidiu com a bola atual. Para fazer isso, abrimos outro loop for para percorrer todas as bolas no array `balls[]`.
- Imediatamente dentro de nosso loop for, usamos uma instrução if para verificar se a bola atual em loop é a mesma bola que estamos verificando no momento. Não queremos verificar se uma bola colidiu consigo mesma! Para fazer isso, verificamos se a bola atual (ou seja, a bola cujo método `collisionDetect` está sendo invocado) é a mesma que a bola de loop (ou seja, a bola que está sendo referenciada pela iteração atual do loop for no `collisionDetect` método). Nós então usamos ! para negar a verificação, para que o código dentro da instrução if seja executado apenas se eles não forem iguais.
- Em seguida, usamos um algoritmo comum para verificar a colisão de dois círculos. Estamos basicamente verificando se alguma das áreas dos dois círculos se sobrepõe. Isso é explicado ainda mais na 2D collision detection.
- Se uma colisão for detectada, o código dentro da instrução if interna será executado. Neste caso, estamos apenas definindo a propriedade color de ambos os círculos para uma nova cor aleatória. Poderíamos ter feito algo muito mais complexo, como fazer com que as bolas saltassem umas das outras de forma realista, mas isso teria sido muito mais complexo de implementar. Para essas simulações físicas, os desenvolvedores tendem a usar jogos ou bibliotecas físicas, como PhysicsJS, matter.js, Phaser, etc.

Você também precisa chamar esse método em cada quadro da animação. Adicione o seguinte abaixo do `balls[i].update();`:

collision

```
balls[i].collisionDetect();
```

Prática

Agora vamos adicionar um círculo maligno controlado pelo usuário, que vai comer as bolas se elas forem capturadas. Também queremos testar suas habilidades de construção de objetos criando um objeto Shape() genérico do qual nossas bolas e círculo maligno podem herdar. Por fim, queremos adicionar um contador de pontuação para rastrear o número de bolas a serem capturadas.

Para exercitar faça a atividade a seguir:

Criando nossos novos objetos

Primeiro de tudo, altere seu construtor `Ball()` existente para que ele se torne um construtor `Shape()` e adicione um novo construtor `Ball()` :

- O construtor `Shape()` deve definir as propriedades `x, y, velX, e velY` da mesma maneira que o construtor `Ball()` fez originalmente, mas não as propriedades de `color` e `size`.
- Também deve definir uma nova propriedade chamada `exists`, que é usada para rastrear se as bolas existem no programa (não foram comidas pelo círculo do mal). Este deve ser um booleano (`true/false`).
- O construtor `Ball()` deve herdar as propriedades `x, y, velX, velY, e exists` do construtor `Shape()`.
- Ele também deve definir uma propriedade `color` e uma `size`, como fez o construtor `Ball()` original.
- Lembre-se de definir o `prototype` e o `constructor` do construtor `Ball()` adequadamente.

As definições do método `ball draw(), update(), e collisionDetect()` devem permanecer exatamente iguais às anteriores.

Você também precisa adicionar um novo parâmetro à nova chamada de construtor `new Ball( ... )` — o parâmetro `exists` deve ser o quinto parâmetro, e deve receber um valor `true`.

Neste ponto, tente recarregar o código — ele deve funcionar da mesma forma que antes, com nossos objetos redesenhados.

Resposta

resposta

```
// define shape constructor

function Shape(x, y, velX, velY, exists) {
    this.x = x;
    this.y = y;
    this.velX = velX;
    this.velY = velY;
    this.exists = exists;
}

// define Ball constructor, inheriting from Shape

function Ball(x, y, velX, velY, exists, color, size) {
    Shape.call(this, x, y, velX, velY, exists);

    this.color = color;
    this.size = size;
}

Ball.prototype = Object.create(Shape.prototype);
Ball.prototype.constructor = Ball;
```

Definindo EvilCircle()

Agora é hora de conhecer o inimigo — o EvilCircle()! Nosso jogo só envolverá um círculo maligno, mas ainda vamos defini-lo usando um construtor que herda de Shape() para lhe dar alguma prática.

- O construtor EvilCircle() deve herdar x, y, velX, velY, e exists de Shape(), mas velX e velY devem sempre ser iguais a 20.
- Ele também deve definir suas próprias propriedades, da seguinte maneira: color — 'white' size — 10

Resposta

resposta

```
// define EvilCircle constructor, inheriting from Shape

function EvilCircle(x, y, exists) {
    Shape.call(this, x, y, 20, 20, exists);

    this.color = 'white';
    this.size = 10;
}

EvilCircle.prototype = Object.create(Shape.prototype);
EvilCircle.prototype.constructor = EvilCircle;
```

Definindo os métodos de EvilCircle()

EvilCircle() deve ter quatro métodos, conforme descrito abaixo:

draw()

Este método tem o mesmo propósito que o método draw() de Ball(): Ele desenha a instância do objeto na tela. Ele funcionará de maneira muito semelhante, portanto, você pode começar copiando a definição Ball.prototype.draw. Você deve então fazer as seguintes alterações:

- Nós queremos que o círculo do mal não seja preenchido, mas apenas tenha uma linha externa (traço). Você pode conseguir isso atualizando fillStyle e fill() para strokeStyle e stroke().
- Também queremos tornar o traço um pouco mais espesso, para que você possa ver o círculo maligno com mais facilidade. Isso pode ser obtido definindo um valor para lineWidth em algum lugar após a chamada beginPath() (3 será suficiente).

Resposta

draw()

```
EvilCircle.prototype.draw = function() {
  ctx.beginPath();
  ctx.strokeStyle = this.color;
  ctx.lineWidth = 3;
  ctx.arc(this.x, this.y, this.size, 0, 2 * Math.PI);
  ctx.stroke();
};
```

checkBounds()

Este método fará a mesma coisa que a primeira parte da função update() do Ball() — olhe para ver se o círculo do mal vai sair da borda da tela, e pare de fazer isso. Novamente, você pode simplesmente copiar a definição de Ball.prototype.update, mas há algumas alterações que você deve fazer:

- Livre-se das duas últimas linhas - não queremos atualizar automaticamente a posição do círculo maligno em todos os quadros, pois estaremos mudando isso de alguma outra forma, como você verá abaixo.
- Dentro das instruções if(), se os testes retornam true, não queremos atualizar o velX/velY; Em vez disso, queremos alterar o valor de x/y para que o círculo maligno seja devolvido na tela um pouco. Adicionar ou subtrair (conforme apropriado) a propriedade size do círculo maligno faria sentido.

Resposta

checkBounds

```
EvilCircle.prototype.checkBounds = function() {
    if((this.x + this.size) >= width) {
        this.x -= this.size;
    }

    if((this.x - this.size) <= 0) {
        this.x += this.size;
    }

    if((this.y + this.size) >= height) {
        this.y -= this.size;
    }

    if((this.y - this.size) <= 0) {
        this.y += this.size;
    }
};
```

setControls()

Esse método adicionará um ouvinte de evento `onkeydown` ao objeto window para que, quando determinadas teclas do teclado forem pressionadas, possamos mover o círculo maligno ao redor. O bloco de código a seguir deve ser colocado dentro da definição do método:

setControls

```
EvilCircle.prototype.setControls = function() {
    var _this = this;
    window.onkeydown = function(e) {
        if(e.key === 'a') {
            _this.x -= _this.velX;
        } else if(e.key === 'd') {
            _this.x += _this.velX;
        } else if(e.key === 'w') {
            _this.y -= _this.velY;
        } else if(e.key === 's') {
            _this.y += _this.velY;
        }
    };
};
```

Assim, quando uma tecla é pressionada, a propriedade keyCode é consultada para ver qual tecla é pressionada. Se for um dos quatro representados pelos códigos de teclas

especificados, o círculo maligno se moverá para a esquerda / direita / para cima / para baixo.

Para um ponto de bônus, deixe-nos saber a quais chaves os códigos de teclas específicos estão mapeados. Para outro ponto de bônus, você pode nos dizer por que precisamos definir var _this = this; na posição em que está? É algo a ver com o escopo da função.

collisionDetect()

- Este método irá agir de forma muito semelhante ao método `collisionDetect()` do `Ball()`, então você pode usar uma cópia disso como base deste novo método. Mas há algumas diferenças:
- Na declaração `if` externa, você não precisa mais verificar se a bola atual na iteração é igual à bola que está fazendo a verificação - porque ela não é mais uma bola, é o círculo do mal! Em vez disso, você precisa fazer um teste para ver se a bola que está sendo checada existe (com qual propriedade você poderia fazer isso?). Se não existe, já foi comido pelo círculo do mal, por isso não há necessidade de verificá-lo novamente. Na instrução if interna, você não quer mais que os objetos mudem de cor quando uma colisão é detectada — em vez disso, você quer definir quaisquer bolas que colidam com o círculo maligno para não existir mais.

collisionDetect

```
EvilCircle.prototype.collisionDetect = function() {
    for(let j = 0; j < balls.length; j++) {
        if( balls[j].exists ) {
            const dx = this.x - balls[j].x;
            const dy = this.y - balls[j].y;
            const distance = Math.sqrt(dx * dx + dy * dy);

            if (distance < this.size + balls[j].size) {
                balls[j].exists = false;
                count--;
                para.textContent = 'Ball count: ' + count;
            }
        }
    }
};
```

Trazendo o círculo do mal para o programa

Agora nós definimos o círculo do mal, precisamos realmente fazer isso aparecer em nossa cena. Para fazer isso, você precisa fazer algumas alterações na função `loop()` .

- Crie 25 bolas

balls

```
    const balls = [];

while(balls.length < 25) {
    const size = random(10,20);
    let ball = new Ball(
        // ball position always drawn at least one ball width
        // away from the adge of the canvas, to avoid drawing errors
        random(0 + size,width - size),
        random(0 + size,height - size),
        random(-7,7),
        random(-7,7),
        true,
        'rgb(' + random(0,255) + ',' + random(0,255) + ',' +
random(0,255) +')',
        size
    );
    balls.push(ball);
}
```

- Primeiro de tudo, crie uma nova instância de objeto do círculo do mal (especificando os parâmetros necessários) e, em seguida, chame seu método `setControls()`. Você só precisa fazer essas duas coisas uma vez, não em todas as iterações do loop.

evil

```
  let evil = new EvilCircle(random(0,width), random(0,height), true);
  evil.setControls();
```

- No ponto em que você percorre todas as bolas e chama as funções `draw(), update(), e collisionDetect()` para cada uma, faça com que essas funções sejam chamadas apenas se a bola atual existir. Chame os métodos `draw(), checkBounds(), e collisionDetect()` da instância do mal ball em cada iteração do loop.

loop

```
  function loop() {
      ctx.fillStyle = 'rgba(0,0,0,0.25)';
      ctx.fillRect(0,0,width,height);

      for(let i = 0; i < balls.length; i++) {
          if(balls[i].exists) {
              balls[i].draw();
              balls[i].update();
              balls[i].collisionDetect();
          }
```

```
        }

        evil.draw();
        evil.checkBounds();
        evil.collisionDetect();

        requestAnimationFrame(loop);
    }
```

Exercício

- Modifique o código criado para utilizar a estrutura de classes do ES6

Info

Fonte: MDN web docs