Capítulo 1 – Introdução ao Canvas

<canvas> é um elemento HTML que pode ser usado para desenhar gráficos, combinar imagens ou criar animações via script (geralmente JavaScript).

Este material descreve como usar o elemento <canvas> para desenhar gráficos 2D, começando com o básico. Os exemplos fornecidos devem fornecer algumas ideias claras sobre o que você pode fazer com a tela e fornecer trechos de código que podem ajudar você a começar a criar seu próprio conteúdo.

O elemento <canvas>

À primeira vista, <canvas> parece com o elemento <img>, com a única diferença de que ele não possui os atributos src alt. De fato, o <canvas> tem apenas dois atributos, width e height. Ambos são opcionais e também podem ser configurados usando as propriedades do DOM . Quando estes atributos não são especificados, a tela terá inicialmente 300 pixels de largura e 150 pixels de altura. O elemento pode ser dimensionado arbitrariamente por CSS , mas durante a renderização, a imagem é dimensionada para se ajustar ao tamanho do layout: se o dimensionamento do CSS não respeitar a proporção da tela inicial, ela aparecerá distorcida.

```
<canvas id="tutorial" width="150" height="150"></canvas>
```

O atributo **id** não é específico ao <canvas>, mas é um dos atributos HTML globais que podem ser aplicados a qualquer elemento HTML (como class, por exemplo). É sempre uma boa ideia fornecer um id, pois isso facilita a identificação em um script.

O <canvas> pode ser estilizado como qualquer elemento (margin, border, background...). Essas regras, no entanto, não afetam o desenho real na tela. Veremos como isso é feito em um capítulo dedicado. Quando nenhuma regra de estilo é aplicada à tela, ela será inicialmente totalmente transparente.

O contexto de renderização

O elemento <canvas> cria uma superfície de desenho de tamanho fixo que expõe um ou mais contextos de renderização, que são usados para criar e manipular o conteúdo mostrado. Neste material, focaremos no contexto de renderização 2D. Outros contextos podem fornecer diferentes tipos de renderização, por exemplo: o WebGL usa um contexto 3D baseado no OpenGL ES .

A tela está inicialmente em branco. Para exibir algo, um script precisa primeiro acessar o contexto de renderização e desenhar nele. O elemento <canvas> possui um método chamado getContext(), usado para obter o contexto de renderização e suas funções de desenho. getContext() usa um parâmetro, o tipo de contexto. Para gráficos 2D, como os abordados neste material, você especifica "2d" para obter um “CanvasRenderingContext2D”.

```
var canvas = document.getElementById('tutorial');
var ctx = canvas.getContext('2d');
```

A primeira linha do script recupera o nó no DOM que representa o elemento <canvas> chamando o método document.getElementById().Depois de ter o nó do elemento, você pode acessar o contexto do desenho usando seu método getContext().

Estrutura inicial

Aqui está um modelo minimalista, que usaremos como ponto de partida para exemplos posteriores. Não é uma boa prática incorporar um script dentro do HTML. Fazemos isso aqui para manter o exemplo conciso.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8"/>
    <script type="text/javascript">
      function desenhar() {
        var canvas = document.getElementById('meucanvas');
        if (canvas.getContext) {
          var ctx = canvas.getContext('2d');
        }
      }
    </script>
    <style type="text/css">
      canvas { border: 1px solid black; }
    </style>
  </head>
  <body onload="desenhar();">
    <canvas id="meucanvas" width="150" height="150"></canvas>
  </body>
</html>
```

O script inclui uma função chamada desenhar(), que é executada quando a página termina de carregar. Isso é feito ouvindo o evento load no documento. Esta função também poderia ser chamada usando window.setTimeout(), window.setInterval() ou qualquer outro manipulador de eventos, desde que a página tenha sido carregada pela primeira vez.

Salve o arquivo o conteúdo acima em um arquivo chamado aula.html e, em seguida, abra no seu navegador. Você verá um quadrado em branco.

Exemplo com dois retângulos

Para começar, vamos dar uma olhada em um exemplo simples que desenha dois retângulos que se cruzam, um dos quais possui transparência alfa. Vamos explorar como isso funciona com mais detalhes nos exemplos posteriores.

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="utf-8"/>
  <script type="application/javascript">
    function desenhar() {
      var canvas = document.getElementById('canvas');
      if (canvas.getContext) {
        var ctx = canvas.getContext('2d');

        ctx.fillStyle = 'rgb(200, 0, 0)';
        ctx.fillRect(10, 10, 50, 50);

        ctx.fillStyle = 'rgba(0, 0, 200, 0.5)';
        ctx.fillRect(30, 30, 50, 50);
      }
    }
  </script>
 </head>
 <body onload="desenhar();">
   <canvas id="canvas" width="150" height="150"></canvas>
 </body>
</html>
```

Salve o código acima e abra no seu navegador, você verá um quadrado azul e um vermelho sobrepostos.

Capítulo 2 – Desenhando formas no canvas

Agora que configuramos nosso ambiente do canvas, podemos entrar em detalhes de como desenhar na tela. No final deste capítulo, você aprenderá a desenhar retângulos, triângulos, linhas, arcos e curvas, proporcionando familiaridade com algumas das formas básicas. Trabalhar com caminhos é essencial ao desenhar objetos na tela e veremos como isso pode ser feito.

A grade na tela

Antes de começarmos a desenhar, precisamos falar sobre a grade da tela ou o espaço de coordenadas. Nosso esqueleto HTML do capítulo anterior tinha um elemento canvas com 150 pixels de largura e 150 pixels de altura.

À baixo, você vê esta tela com a grade padrão sobreposta. Normalmente 1 unidade na grade corresponde a 1 pixel na tela. A origem dessa grade é posicionada no canto superior esquerdo na coordenada (0,0). Todos os elementos são colocados em relação a essa origem.

Portanto, a posição do canto superior esquerdo do quadrado azul se torna x pixels da esquerda e y pixels do topo, na coordenada (x, y). Posteriormente neste material, veremos como podemos traduzir a origem para uma posição diferente, girar a grade e até escalá-la, mas por enquanto vamos nos ater ao padrão.

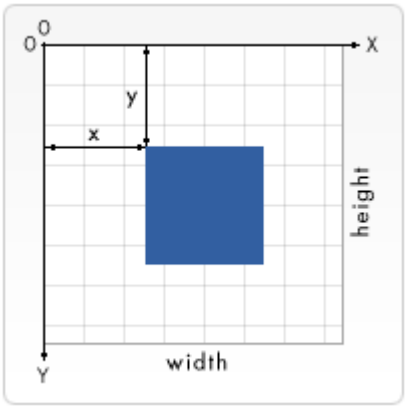


Figura 1 - Eixos x e y na tela

2.1 Desenhando Retângulos

O elemento <canvas> suporta apenas duas formas primitivas: retângulos e caminhos (listas de pontos conectados por linhas). Todas as outras formas devem ser criadas combinando um ou mais caminhos. Felizmente, temos uma variedade de funções de desenho de caminho que permitem compor formas muito complexas.

Primeiro, vamos olhar para o retângulo. Existem três funções que desenham retângulos na tela:

fillRect(x, y, width, height) ➔ Desenha um retângulo preenchido.

strokeRect(x, y, width, height) ➔ Desenha um contorno retangular.

clearRect(x, y, width, height) ➔ Limpa a área retangular especificada, tornando-a totalmente transparente.

Cada uma dessas três funções usa os mesmos parâmetros: x e y especificam a posição na tela (em relação à origem) do canto superior esquerdo do retângulo. Já o width e o height fornecem o tamanho do retângulo.

Abaixo está a função desenhar() do capítulo anterior, mas agora ela está fazendo uso dessas três funções.

```
function desenhar() {
  var canvas = document.getElementById('canvas');
  if (canvas.getContext) {
    var ctx = canvas.getContext('2d');

    ctx.fillRect(25, 25, 100, 100);
    ctx.clearRect(45, 45, 60, 60);
    ctx.strokeRect(50, 50, 50, 50);
  }
}
```

A função fillRect() desenha um quadrado preto grande de 100 pixels em cada lado. A função clearRect() apaga um quadrado de 60x60 pixels do centro e depois strokeRect() é chamada para criar um contorno retangular de 50x50 pixels dentro do quadrado limpo.

Nas próximas páginas, veremos dois métodos alternativos clearRect() e também veremos como alterar o estilo de cor e traçado das formas renderizadas.

Diferente das funções de caminho que veremos na próxima seção, todas as três funções retangulares são desenhadas imediatamente para a tela.

2.2 Traçando caminhos

Agora vamos ver os caminhos. Um caminho é uma lista de pontos, conectados por segmentos de linhas que podem ser de diferentes formas, curvas ou não, de largura e cor diferentes. Um caminho, ou mesmo um subcaminho, pode ser fechado. Para criar formas usando caminhos, tomamos algumas etapas extras:

1. Primeiro, você cria o caminho.
2. Então você usa comandos de desenho para desenhar no caminho.
3. Após a criação do caminho, você pode traçar ou preencher o caminho para renderizá-lo.

Aqui estão as funções usadas para executar estas etapas:

beginPath() ➔ Cria um novo caminho. Uma vez criados, os futuros comandos de desenho são direcionados para o caminho e usados para construí-lo.

closePath() ➔ Adiciona uma linha reta ao caminho, indo para o início do subcaminho atual.

stroke() ➔ Desenha a forma definindo seu contorno.

fill() ➔ Desenha uma forma sólida preenchendo a área de conteúdo do caminho.

O primeiro passo para criar um caminho é chamar o beginPath(). Internamente, os caminhos são armazenados como uma lista de subcaminhos (linhas, arcos etc.) que juntos formam uma forma. Sempre que esse método é chamado, a lista é redefinida e podemos começar a desenhar novas formas.

A segunda etapa é chamar os métodos que realmente especificam os caminhos a serem desenhados. Veremos isso em breve.

O terceiro, e uma etapa opcional, é chamar closePath(). Este método tenta fechar a forma desenhando uma linha reta do ponto atual até o início. Se a forma já foi fechada ou há apenas um ponto na lista, essa função não faz nada.

2.3 Desenhando um triângulo

Por exemplo, o código para desenhar um triângulo seria mais ou menos assim:

```
function desenhar() {
  var canvas = document.getElementById('canvas');
  if (canvas.getContext) {
    var ctx = canvas.getContext('2d');

    ctx.beginPath();
    ctx.moveTo(75, 50);
    ctx.lineTo(100, 75);
    ctx.lineTo(100, 25);
    ctx.fill();
  }
}
```

2.3 Mover a caneta

Uma função muito útil, que na verdade não desenha nada, mas se torna parte da lista de caminhos descrita acima, é a fução moveTo(). Você provavelmente pode pensar nisso como levantar uma caneta ou lápis de um ponto em um pedaço de papel e colocá-lo no outro.

moveTo(x, y) ➔ Move a caneta para as coordenadas especificadas por x e y.

Quando a tela é inicializada ou a função beginPath() é chamada, normalmente você deseja usar a função moveTo() para colocar o ponto de partida em outro lugar. Também poderíamos usar moveTo() para desenhar caminhos não conectados. Dê uma olhada no rosto sorridente da função abaixo. A função arc() ainda não foi vista, mas será vista logo mais.

Para tentar você mesmo, você pode usar o trecho de código abaixo. Basta colá-lo na função desenhar() que vimos anteriormente:

```
function desenhar() {
  var canvas = document.getElementById('canvas');
  if (canvas.getContext) {
     var ctx = canvas.getContext('2d');

    ctx.beginPath();
    ctx.arc(75, 75, 50, 0, Math.PI * 2, true);
    ctx.moveTo(110, 75);
    ctx.arc(75, 75, 35, 0, Math.PI, false);
    ctx.moveTo(65, 65);
    ctx.arc(60, 65, 5, 0, Math.PI * 2, true);
    ctx.moveTo(95, 65);
    ctx.arc(90, 65, 5, 0, Math.PI * 2, true);
    ctx.stroke();
  }
}
```

2.4 Linhas

Para desenhar linhas retas, use o método lineTo():

lineTo(x, y) ➔ Desenha uma linha da posição atual do desenho para a posição especificada por x e y.

Este método usa dois argumentos x e y, os quais são as coordenadas do ponto final da linha. O ponto de partida depende dos caminhos desenhados anteriormente, onde o ponto final do caminho anterior é o ponto inicial para o seguinte, etc. O ponto inicial também pode ser alterado usando o método moveTo().

O exemplo a seguir desenha dois triângulos, um preenchido e outro delineado.

```
function desenhar() {
  var canvas = document.getElementById('canvas');
  if (canvas.getContext) {
    var ctx = canvas.getContext('2d');

    // Triangulo com Fill
    ctx.beginPath();
    ctx.moveTo(25, 25);
    ctx.lineTo(105, 25);
    ctx.lineTo(25, 105);
    ctx.fill();

    // Triangulo com stroke
    ctx.beginPath();
    ctx.moveTo(125, 125);
    ctx.lineTo(125, 45);
    ctx.lineTo(45, 125);
    ctx.closePath();
    ctx.stroke();
  }
}
```

Esta função começa chamando beginPath() para iniciar um novo caminho de forma. Em seguida, usamos o método moveTo() para mover o ponto inicial para a posição desejada.

Realize o teste deste código. Você notará a diferença entre o triângulo preenchido e o traçado. Isso ocorre, como mencionado acima, porque as formas são fechadas automaticamente quando um caminho é preenchido, mas não quando são traçadas. Se deixássemos de fora o closePath() do triângulo traçado, apenas duas linhas teriam sido desenhadas, não um triângulo completo.

2.5 Arcos

Para desenhar arcos ou círculos, usamos os métodos arc() ou arcTo().

arc(x, y, raio, inicioAngulo, fimAngulo, antihorario) ➔ Desenha um arco centralizado na posição (x, y) com o raio r começando em inicioAngulo e terminando em fimAngulo indo na direção indicada (horário ou anti-horário).

arcTo(x1, y1, x2, y2, raio) ➔ Desenha um arco com os pontos e o raio de controle fornecidos, conectados ao ponto anterior por uma linha reta.

Vamos dar uma olhada mais detalhada no método arc(), que utiliza seis parâmetros: x e y são as coordenadas do centro do círculo em que o arco deve ser desenhado. Raio é autoexplicativo. Os parâmetros inicioAngulo e fimAngulo definem os pontos inicial e final do arco em radianos, ao longo da curva do círculo. Estes são medidos a partir do eixo x. O o parâmetro anti-horário é um valor booleano que, quando **true**, desenha o arco no sentido anti-horário; caso contrário, o arco é desenhado no sentido horário.

O exemplo a seguir é um pouco mais complexo do que os que vimos anteriormente. Ele desenha 12 arcos diferentes, todos com ângulos e preenchimentos diferentes.

Os dois loops for são para percorrer as linhas e colunas dos arcos. Para cada arco, iniciamos um novo caminho chamando beginPath(). No código, cada um dos parâmetros do arco está em uma variável para maior clareza, mas você não necessariamente precisaria fazer isso na vida real.

As coordenadas x e y devem ser suficientemente claras. Raio e inicioAngulo são fixos. O parâmetro fimAngulo começa em 180 graus (metade de um círculo), na primeira coluna e é aumentada em etapas de 90 graus, culminando em um círculo completo na última coluna.

A instrução para o parâmetro anti-horário faz com que a primeira e a terceira linha sejam desenhadas como arcos no sentido horário e a segunda e quarta linha como arcos no sentido anti-horário.

```
function desenhar() {
  var canvas = document.getElementById('canvas');
  if (canvas.getContext) {
    var ctx = canvas.getContext('2d');

    for (var i = 0; i < 4; i++) {
      for (var j = 0; j < 3; j++) {
        ctx.beginPath();
        var x = 25 + j * 50; // coordenada x
        var y = 25 + i * 50; // coordenada y
        var raio = 20; // raio
        var inicioAngulo = 0;
        var fimAngulo = Math.PI + (Math.PI * j) / 2;
        var antiHoraio = i % 2 !== 0; //horário ou anti-horário
        ctx.arc(x, y, raio, inicioAngulo, fimAngulo,
antiHorario);
        if (i > 1) {
          ctx.fill();
        } else {
          ctx.stroke();
        }
      }
    }
  }
}
```

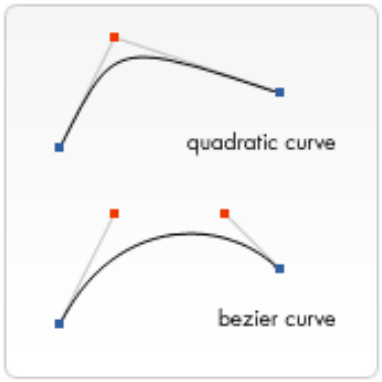
2.6 Bezier e Curvas Quadráticas

O próximo tipo de caminho disponível são as curvas de Bézier, disponíveis nas variações cúbica e quadrática. Estes são geralmente usados para desenhar formas orgânicas complexas.

quadraticCurveTo(cp1x, cp1y, x, y) ➔ Desenha uma curva quadrática de Bézier da posição atual da caneta até o ponto final especificado por x e y, usando o ponto de controle especificado por cp1x e cp1y.

bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y) ➔ Desenha uma curva cúbica de Bézier da posição atual da caneta até o ponto final especificado por x e y, usando os pontos de controle especificados por (cp1x, cp1y) e (cp2x, cp2y).

A diferença entre elas pode ser melhor descrita usando a imagem a seguir. Uma curva quadrática de Bézier tem um ponto inicial e final (pontos azuis) e apenas um ponto de controle (indicado pelo ponto vermelho) enquanto uma curva cúbica de Bézier usa dois pontos de controle.



Os parâmetros x e y em ambos os métodos são as coordenadas do ponto final. Já cp1x e cp1y são as coordenadas do primeiro ponto de controle, cp2x e cp2y são as coordenadas do segundo ponto de controle.

O uso de curvas quadráticas e cúbicas de Bézier pode ser bastante desafiador, porque, diferentemente do software de desenho vetorial como o Adobe Illustrator, não temos feedback visual direto sobre o que estamos fazendo. Isso torna bastante difícil desenhar formas complexas. No exemplo a seguir, desenharemos algumas formas orgânicas simples, mas se você tiver tempo e, acima de tudo, paciência, podem ser criadas formas muito mais complexas.

Não há nada muito difícil nesses exemplos. Nos dois casos, vemos uma sucessão de curvas sendo desenhadas, que finalmente resultam em uma forma completa.

Neste primeiro exemplo, nós desenhamos um balão de fala utilizando as curvas quadráticas de Bézier:

```
function desenhar() {
  var canvas = document.getElementById('canvas');
  if (canvas.getContext) {
    var ctx = canvas.getContext('2d');

    // Exemplo de curvas quadráticas
    ctx.beginPath();
    ctx.moveTo(75, 25);
```

```
    ctx.quadraticCurveTo(25, 25, 25, 62.5);
    ctx.quadraticCurveTo(25, 100, 50, 100);
    ctx.quadraticCurveTo(50, 120, 30, 125);
    ctx.quadraticCurveTo(60, 120, 65, 100);
    ctx.quadraticCurveTo(125, 100, 125, 62.5);
    ctx.quadraticCurveTo(125, 25, 75, 25);
    ctx.stroke();
  }
}
```

Neste segundo exemplo, vamos desenhar um coração usando curvas cúbicas de Bézier:

```
function desenhar() {
  var canvas = document.getElementById('canvas');
  if (canvas.getContext) {
    var ctx = canvas.getContext('2d');

    // Exemplo de curvas cúbicas
    ctx.beginPath();
    ctx.moveTo(75, 40);
    ctx.bezierCurveTo(75, 37, 70, 25, 50, 25);
    ctx.bezierCurveTo(20, 25, 20, 62.5, 20, 62.5);
    ctx.bezierCurveTo(20, 80, 40, 102, 75, 120);
    ctx.bezierCurveTo(110, 102, 130, 80, 130, 62.5);
    ctx.bezierCurveTo(130, 62.5, 130, 25, 100, 25);
    ctx.bezierCurveTo(85, 25, 75, 37, 75, 40);
    ctx.fill();
  }
}
```

2.7 Retângulos

Além dos três métodos que vimos em Desenhando retângulos, que desenham formas retangulares diretamente na tela, há também o método rect(), que adiciona um caminho retangular a um caminho aberto no momento.

rect(x, y, largura, altura) ➔ Desenha um retângulo cujo canto superior esquerdo é especificado por (x, y) com as dimensões de largura e altura.

Antes que esse método seja executado, o método moveTo() é chamado automaticamente com os parâmetros (x, y). Em outras palavras, a posição atual da caneta é redefinida automaticamente para as coordenadas padrão.

2.7 Fazendo combinações

Até o momento, cada exemplo nesta página usou apenas um tipo de função de caminho por forma. No entanto, não há limitação para o número ou tipos de caminhos que você pode usar para criar uma forma. Portanto, neste exemplo final, vamos combinar todas as funções de caminho para criar um conjunto de personagens de um jogo muito famoso:

```
function desenhar() {
  var canvas = document.getElementById('canvas');
  if (canvas.getContext) {
    var ctx = canvas.getContext('2d');

    roundedRect(ctx, 12, 12, 150, 150, 15);
    roundedRect(ctx, 19, 19, 150, 150, 9);
    roundedRect(ctx, 53, 53, 49, 33, 10);
    roundedRect(ctx, 53, 119, 49, 16, 6);
    roundedRect(ctx, 135, 53, 49, 33, 10);
    roundedRect(ctx, 135, 119, 25, 49, 10);

    ctx.beginPath();
    ctx.arc(37, 37, 13, Math.PI / 7, -Math.PI / 7, false);
    ctx.lineTo(31, 37);
    ctx.fill();

    for (var i = 0; i < 8; i++) {
      ctx.fillRect(51 + i * 16, 35, 4, 4);
    }

    for (i = 0; i < 6; i++) {
      ctx.fillRect(115, 51 + i * 16, 4, 4);
    }

    for (i = 0; i < 8; i++) {
      ctx.fillRect(51 + i * 16, 99, 4, 4);
    }

    ctx.beginPath();
    ctx.moveTo(83, 116);
    ctx.lineTo(83, 102);
    ctx.bezierCurveTo(83, 94, 89, 88, 97, 88);
    ctx.bezierCurveTo(105, 88, 111, 94, 111, 102);
    ctx.lineTo(111, 116);
    ctx.lineTo(106.333, 111.333);
    ctx.lineTo(101.666, 116);
    ctx.lineTo(97, 111.333);
    ctx.lineTo(92.333, 116);
    ctx.lineTo(87.666, 111.333);
    ctx.lineTo(83, 116);
    ctx.fill();
```

```
    ctx.fillStyle = 'white';
    ctx.beginPath();
    ctx.moveTo(91, 96);
    ctx.bezierCurveTo(88, 96, 87, 99, 87, 101);
    ctx.bezierCurveTo(87, 103, 88, 106, 91, 106);
    ctx.bezierCurveTo(94, 106, 95, 103, 95, 101);
    ctx.bezierCurveTo(95, 99, 94, 96, 91, 96);
    ctx.moveTo(103, 96);
    ctx.bezierCurveTo(100, 96, 99, 99, 99, 101);
    ctx.bezierCurveTo(99, 103, 100, 106, 103, 106);
    ctx.bezierCurveTo(106, 106, 107, 103, 107, 101);
    ctx.bezierCurveTo(107, 99, 106, 96, 103, 96);
    ctx.fill();

    ctx.fillStyle = 'black';
    ctx.beginPath();
    ctx.arc(101, 102, 2, 0, Math.PI * 2, true);
    ctx.fill();

    ctx.beginPath();
    ctx.arc(89, 102, 2, 0, Math.PI * 2, true);
    ctx.fill();
  }
}

// desenhar um retângulo com cantos arredondados.

function roundedRect(ctx, x, y, width, height, radius) {
  ctx.beginPath();
  ctx.moveTo(x, y + radius);
  ctx.lineTo(x, y + height - radius);
  ctx.arcTo(x, y + height, x + radius, y + height, radius);
  ctx.lineTo(x + width - radius, y + height);
  ctx.arcTo(x + width, y + height, x + width, y + height-radius,
radius);
  ctx.lineTo(x + width, y + radius);
  ctx.arcTo(x + width, y, x + width - radius, y, radius);
  ctx.lineTo(x + radius, y);
  ctx.arcTo(x, y, x, y + radius, radius);
  ctx.stroke();
}
```

Não analisaremos isso em detalhes. Porém, as coisas mais importantes a serem observadas são o uso da propriedade fillStyle no contexto do desenho e o uso de uma função utilitária (neste caso roundedRect()). O uso de funções utilitárias para bits de desenho que você faz com frequência pode ser muito útil e reduz a quantidade de código necessária, bem como sua complexidade.

Examinaremos fillStyle mais detalhadamente mais adiante neste material. Aqui, tudo o que estamos fazendo é usá-lo para alterar a cor de preenchimento dos caminhos da cor padrão preto para branco e depois voltar.

2.8 Objetos Path2D

Como vimos no último exemplo, pode haver uma série de caminhos e comandos de desenho para desenhar objetos em seu canvas. Para simplificar o código e melhorar o desempenho, o objeto Path2D, disponível nas versões recentes dos navegadores, permite armazenar em cache ou gravar esses comandos de desenho para uso posterior. Você é capaz de reproduzir seus caminhos rapidamente.

Vamos ver como podemos construir um Path2Dobjeto:

Path2D() ➔ O construtor Path2D retorna um objeto Path2D recém-instanciado, opcionalmente com outro caminho como argumento (ou seja, cria uma cópia) ou, opcionalmente, com uma sequência que consiste em dados do caminho SVG.

```
new Path2D();     // caminho vazio do objeto
new Path2D(path); // cópia de outro objeto Path2D
new Path2D(d);    // caminho para um dado SVG
```

Todos os métodos de caminho como moveTo, rect, arc ou quadraticCurveTo, etc., que nós conhecemos neste capítulo, estão disponíveis em objetos Path2D.

2.8.1 Exemplo

Neste exemplo, estamos criando um retângulo e um círculo. Ambos são armazenados como um objeto Path2D, para que estejam disponíveis para uso posterior. Com a nova API Path2D, vários métodos foram atualizados para aceitar opcionalmente um objeto Path2D a ser usado em vez do caminho atual. Aqui, stroke e fill são usados com um argumento de caminho para desenhar os dois objetos na tela, por exemplo.

```
function draw() {
  var canvas = document.getElementById('canvas');
  if (canvas.getContext) {
    var ctx = canvas.getContext('2d');

    var retangulo = new Path2D();
    retangulo.rect(10, 10, 50, 50);

    var circulo = new Path2D();
    circulo.moveTo(125, 35);
    circulo.arc(100, 35, 25, 0, 2 * Math.PI);
```

```
    ctx.stroke(retangulo);
    ctx.fill(circulo);
  }
}
```

Capítulo 3 – Usando imagens

Até agora, criamos nossas próprias formas. Um dos recursos mais interessantes do <canvas> é a capacidade de usar imagens. Eles podem ser usados para fazer composição dinâmica de fotos ou como pano de fundo de gráficos, para sprites em jogos e assim por diante. Imagens externas podem ser usadas em qualquer formato suportado pelo navegador, como PNG, GIF ou JPEG.

A importação de imagens em uma tela é basicamente um processo de duas etapas:

1. Obtenha uma referência a um objeto HTMLImageElement ou a outro elemento da tela como fonte. Também é possível usar imagens fornecendo uma URL.
2. Desenhe a imagem na tela usando a função drawImage()

Vamos dar uma olhada em como fazer isso.

3.1 Desenhando imagens

O canvas pode usar qualquer um dos seguintes tipos de dados como fonte de imagem:

HTMLImageElement ➔ Estas são imagens criadas usando o construtor Image(), bem como qualquer elemento <img>.

SVGImageElement ➔ Estas são imagens incorporadas usando o elemento <image>.

HTMLVideoElement ➔ O uso de um elemento HTML <video> como fonte de imagem captura o quadro atual do vídeo e o usa como imagem.

HTMLCanvasElement ➔ Você pode usar outro elemento <canvas> como sua fonte de imagem.

Essas fontes são coletivamente referidas pelo tipo **CanvasImageSource**.

Existem várias maneiras de obter imagens para uso em uma tela, vamos conhece-lás.

3.2 Criando uma imagem do zero

Uma opção é criar novos objetos HTMLImageElement em nosso script. Para fazer isso, você pode usar o construtor Image():

```
var img = new Image();   // cria novo elemento img
img.src = 'minhaimagem.png'; // indica o caminho da imagem
```

Quando esse script é executado, a imagem começa a carregar.

Se você tentar executar drawImage() antes que a imagem termine de carregar, ela não fará nada (ou, em navegadores antigos, pode gerar uma exceção). Portanto, você deve certificar-se de usar o evento load para não tentar isso antes que a imagem seja carregada:

```
var img = new Image();   // cria novo elemento img
img.addEventListener('load', function() {
  // executa drawImage aqui
}, false);
img.src = minhaimagem.png'; // indica o caminho da imagem
```

Se você estiver usando apenas uma imagem externa, pode ser uma boa abordagem, mas depois que precisar rastrear mais de uma, precisamos recorrer a algo mais elaborado.

3.3 Incorporando imagem via dados URL

Outra maneira possível de incluir imagens é através dos dados: url. Os URLs de dados permitem definir completamente uma imagem como uma sequência de caracteres codificados em Base 64 diretamente no seu código.

```
var img = new Image();   // cria novo elemento img
img.src = 'urldaimagem';
```

Uma vantagem dos URLs de dados é que a imagem resultante está disponível imediatamente sem outra ida e volta ao servidor. Outra vantagem potencial é que também é possível encapsular em um arquivo todos os seus CSS , JavaScript , HTML e imagens, tornando-o mais portátil para outros locais.

Algumas desvantagens desse método são que sua imagem não é armazenada em cache e, para imagens maiores, o URL codificado pode se tornar bastante longo.

3.3 Desenhando a imagem no canvas

Depois de termos uma referência ao nosso objeto de imagem de origem, podemos usar o método drawImage() para renderizá-lo na tela. Como veremos mais adiante, o método drawImage() está sobrecarregado e possui várias variantes. Na sua forma mais básica, fica assim:

drawImage(imagem, x, y) ➔ Desenha o CanvasImageSource especificado pelo parâmetro **imagem** nas coordenadas (x, y).

3.3.1 Dimensionamento

A segunda variante do método drawImage()adiciona dois novos parâmetros e permite colocar imagens em escala na tela.

drawImage(image, x, y, width, height) ➔ Isso adiciona os parâmetros largura e altura, que indicam o tamanho para dimensionar a imagem ao desenhá-la na tela.

3.4 Exemplo

Neste exemplo, usaremos uma imagem como papel de parede e repetiremos várias vezes no canvas. Isso é feito simplesmente fazendo um loop e colocando as imagens em escala em diferentes posições. No código abaixo, o primeiro loop itera sobre as linhas. O segundo loop itera sobre as colunas. A imagem é dimensionada, pois seu tamanho real é 64x32 pixels.

```
function draw() {
  var ctx = document.getElementById('canvas').getContext('2d');
  var img = new Image();
  img.onload = function() {
    for (var i = 0; i < 4; i++) {
      for (var j = 0; j < 3; j++) {
        ctx.drawImage(img, j * 50, i * 38, 50, 38);
      }
    }
  };
  img.src = 'http://arieldias.com/novo/material/2019-
2/DJW/img/ovni.png';
}
```

3.4 Fatiamento

A terceira e última variante do método drawImage() possui oito parâmetros, além da fonte da imagem. Permite recortar uma seção da imagem de origem, dimensioná-la e desenhá-la em nossa tela.

drawImage(imagem, sx, sy, sWidth, sHeight, dx, dy, dWidth, dHeight) ➔

Esta função leva a área da imagem de origem especificado pelo retângulo cuja parte superior esquerda canto é (sx, sy) e cuja largura e altura são sWidth e sHeight e desenha-o na tela, colocando-o sobre a tela em (dx, dy) e recortando para o tamanho especificado por dWidth e dHeight. Veja a figura 2 a seguir com esta sequência de coordenadas.

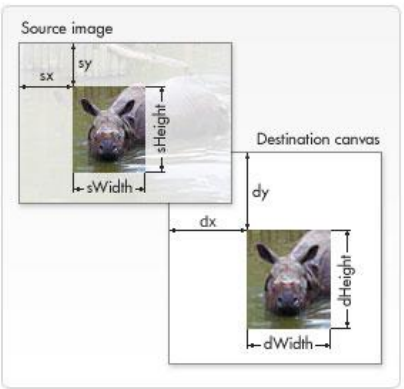


Figura 2 - Recortando a imagem: exemplificando as coordenadas de drawImage

Para realmente entender o que isso faz, pode ser útil olhar as figuras 3 e 4 a seguir. Os quatro primeiros parâmetros definem o local e o tamanho da fatia na imagem de origem. Os últimos quatro parâmetros definem o retângulo no qual desenhar a imagem na tela de destino.

Figura 3 - Imagem Sonic desenhada no canvas não fatiada

Figura 4 - Imagem do Sonic carregada no canvas fatiada

O fatiamento pode ser uma ferramenta útil quando você deseja fazer composições. Você pode ter todos os elementos em um único arquivo de imagem e usar esse método para compor um desenho completo. Por exemplo, se você quiser criar um gráfico, poderá ter uma imagem PNG contendo todo o texto necessário em um único arquivo e, dependendo dos seus dados, poderá alterar a escala do gráfico com bastante facilidade. Outra vantagem é que você não precisa carregar todas as imagens individualmente, o que pode melhorar o desempenho do carregamento.

3.4.1 Exemplo

Neste exemplo, usaremos o mesmo Sonic que no exemplo anterior e recortaremos parte do seu corpo (como feito na figura 3).

```
function desenhar() {
  var canvas = document.getElementById('canvas');
  var ctx = canvas.getContext('2d');

  var img = new Image();
  img.src = 'sonic.jpeg';

  img.onload = function() {
       context.drawImage(img,
                          310, 0, 360, 265, // Área de recorte
(clipping) xorigem, yorigem, larguraorigem, alturaorigem
                   0, 0, 360, 265 // Desenha no canvas xdestino,
ydestino, larguradestino, alturadestino
                               )
                     }
}
```

Referências:

MOZILLA. **MDN WebDocs**. Disponível em: <https://developer.mozilla.org/>. Acesso em: 05/03/2020.