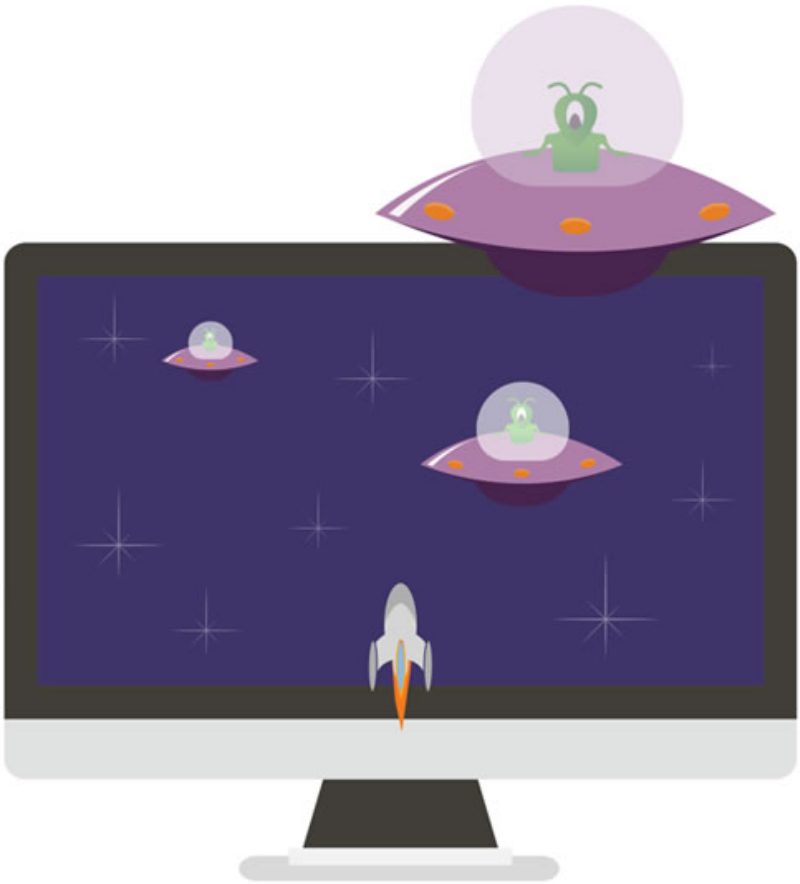
Desenvolva jogos com

HTML5 Canvas e JavaScript



Casa do Código

ÉDERSON CÁSSIO

Introdução

Se você chegou até este livro, é porque está interessado em desenvolver jogos. Aqui focarei no desenvolvimento de jogos para a web, usando a tecnologia **Canvas**, presente na especificação do HTML5 e suportada por todos os maiores browsers modernos, em suas versões mais atualizadas. Pretendo mostrar que desenvolver jogos é uma tarefa na realidade simples, e que não exige grandes curvas de aprendizados com frameworks monstruosos — o básico oferecido pelo ambiente do navegador é o suficiente para começar!

Tudo o que você precisará é de um browser atualizado e um bom editor de textos. Em ambiente Windows, é muito comum o uso do *Notepad++* (http://notepad-plus-plus.org) . Caso você use Linux, é bem provável que sua distribuição já venha com um editor que suporte coloração de sintaxe para várias linguagens (GEdit, KWrite, etc.). No entanto, eu recomendo veementemente o uso do *Brackets* (http://brackets.io) , que foi onde eu criei os códigos. É multiplataforma (funciona em Windows, Linux, Mac OS X) e realiza autocompletar em código JavaScript de forma bastante eficiente.

O jogo que você vai criar já está disponível na web. Você pode jogá-lo a qualquer momento em:

http://gamecursos.com.br/livro/jogo

Estarei sempre presente e atuante no seguinte grupo do Google Groups, como forma de interagir com meus leitores:

http://groups.google.com/forum/#!forum/livro-jogos-html5-canvas

Por fim, todos os códigos e imagens estão disponíveis em (embora eu aconselhe você a *digitar* os códigos e usar os prontos no download apenas para referência em caso de dúvidas):

http://github.com/EdyKnopfler/games-js/archive/master.zip

Bom estudo!

Sumário

1 Fundamentos **1**

1.1 Introdução ao HTML5 Canvas 3
1.2 Começando a desenhar 6
1.3 Animações com *requestAnimationFrame* 22
1.4 Orientação a objetos com JavaScript 26

2 O loop de animação **35**

2.1 Introdução e sprites 35
2.2 Teste para a classe Animacao 38
2.3 Desenvolva a classe Animacao 41
2.4 Implemente a classe Bola 45

3 A interação com o jogador — leitura apurada do teclado **49**

3.1 EventListeners e os eventos keydown e keyup 50
3.2 Detectando se uma tecla está ou não pressionada 55
3.3 Efetuando disparos — detectando somente o primeiro keydown 58

4 Folhas de sprites — spritesheets **69**

4.1 Conceito e abordagem utilizada 69
4.2 Carregando imagens e fazendo recortes (clipping) 72
4.3 Animações de sprite — a classe Spritesheet 75
4.4 Controle o herói pelo teclado e veja sua animação 80

5 Detecção de colisões **87**
5.1 Colisão entre retângulos . . . 87
5.2 Teste da classe Colisor . . . 90
5.3 A classe Colisor . . . 93
5.4 Criando um sprite colidível . . . 95
5.5 Melhorando o código . . . 98

6 Iniciando o desenvolvimento do jogo **107**
6.1 Animação de fundo com efeito parallax . . . 107
6.2 Controle da nave na horizontal e na vertical . . . 118
6.3 Efetuando disparos . . . 123

7 Criando inimigos **129**
7.1 Primeiro teste com nave e inimigos . . . 130
7.2 A classe Ovni . . . 134
7.3 Adicionando fundo em parallax . . . 135
7.4 Adicionando colisão . . . 139
7.5 Estamos experimentando lentidão! . . . 145
7.6 Excluindo os objetos desnecessários . . . 151

8 Incorpore animações, sons, pausa e vidas extras ao jogo **155**
8.1 Organizando o código . . . 156
8.2 Animação cronometrada . . . 162
8.3 Animando a nave com spritesheets . . . 165
8.4 Criando explosões . . . 169
8.5 Pausando o jogo . . . 175
8.6 Sons e música de fundo . . . 179
8.7 Tela de loading . . . 181
8.8 Vidas extras . . . 186
8.9 Pontuação (*score*) . . . 190
8.10 Tela de Game Over . . . 192

9 Publique seu jogo e torne-o conhecido **197**

9.1 Hospede-o em um serviço gratuito 197

9.2 Linkando com as redes sociais 203

Bibliografia **213**

Capítulo 1

Fundamentos

Aqui começa uma fascinante jornada pelos segredos de uma tecnologia que, na verdade, não possui segredo algum. Trata-se do **Canvas**, uma das maravilhas do HTML5.

O Canvas é uma área retangular em uma página web onde podemos criar desenhos programaticamente, usando JavaScript (a linguagem de programação normal das páginas HTML). Com esta tecnologia, podemos criar trabalhos artísticos, animações e jogos, que é o assunto central deste livro.

Com o Canvas, ao longo dos capítulos, iremos desenvolver o jogo da figura 1.1:

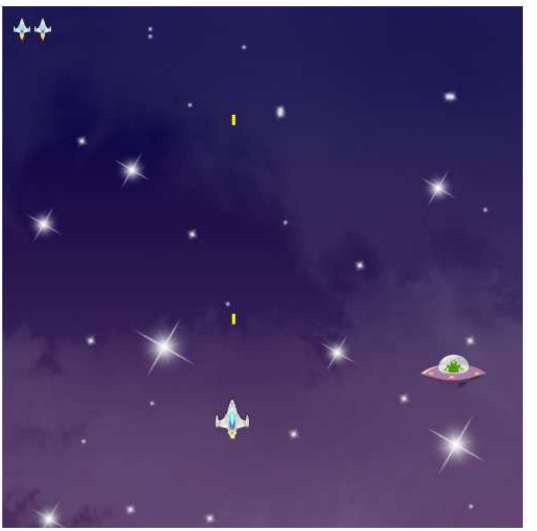


Figura 1.1: Jogo desenvolvido ao longo do livro

Este livro é composto pelos seguintes capítulos:

1. *Fundamentos:* neste capítulo, aprenda como funciona o Canvas, como criar animações via programação e também noções de Orientação a objetos em JavaScript, para que nossos códigos se tornem mais organizados e reaproveitáveis;

2. *O loop de animação:* controle a animação de seus jogos de forma eficiente. Conheça o conceito de *sprites* e aprenda a gerenciá-los em uma animação;

3. *A interação com o jogador — leitura apurada do teclado:* saiba como capturar eventos do teclado de maneira eficiente para jogos;

4. *Folhas de sprites — spritesheets:* anime os elementos de seu jogo individualmente usando imagens contendo todos os quadros de uma animação;

5. *Detecção de colisões:* aprenda a detectar quando os elementos de um jogo "tocam" uns aos outros na tela e execute as ações adequadas;

6. *Iniciando o desenvolvimento do jogo:* um joguinho de nave começará a tomar forma usando todos os elementos aprendidos até então;

7. *Criando inimigos:* adicione verdadeira emoção ao seu jogo, dando

ao herói alguém a quem enfrentar. Usaremos intensivamente a detecção de colisões;

8. *Incorpore animações, sons, pausa e vidas extras ao jogo:* com todos os conceitos aprendidos e bem fixados, você verá como é fácil estender o jogo e adicionar novos elementos. Ao fim do capítulo, você terá sugestões de melhorias que você mesmo poderá tentar realizar, como exercício;

9. *Publique seu jogo e torne-o conhecido:* um passo a passo de como publicar seu jogo na web e divulgá-lo nas redes sociais.

Importante: preparei um pacote de arquivos contendo todos os códigos, imagens e sons utilizados. Em cada novo arquivo que criarmos, indicarei o nome do respectivo arquivo nesse pacote. Realize seu download no endereço:

http://github.com/EdyKnopfler/games-js/archive/master.zip

Antes de começarmos a desenvolver um jogo em específico, é importante nos habituarmos a algumas funções da tecnologia Canvas. O que está esperando? Vamos começar o aprendizado!

1.1 Introdução ao HTML5 Canvas

Para criar um Canvas em uma página HTML, utilizamos a tag `<canvas>`. Os atributos `width` e `height` informam a largura e a altura, respectivamente, da área de desenho. É importante também informar um `id` para podermos trabalhar com ele no código JavaScript:

```
<canvas id="nome_canvas" width="largura" height="altura">
</canvas>
```

Entre as tags de abertura e fechamento, podemos colocar alguma mensagem indicando que o browser não suporta essa tecnologia. Caso o browser a suporte, esse conteúdo é ignorado:

```
<canvas id="meu_canvas" width="300" height="300">
   Seu navegador não suporta o Canvas do HTML5. <br>
   Procure atualizá-lo.
</canvas>
```

Os atributos `width` e `height` da tag `<canvas>` são obrigatórios, pois são os valores usados na geração da imagem. O Canvas pode receber dimen-

sões diferentes via CSS, no entanto, seu processamento sempre será feito usando as dimensões informadas na tag. Se as dimensões no CSS forem diferentes, o browser amplia ou reduz a imagem gerada para deixá-la de acordo com a folha de estilo.

Dado um Canvas com dimensões 100x100 pixels:

```
<canvas id="meu_canvas" width="100" height="100"></canvas>
```

A seguinte formatação CSS fará a imagem ser ampliada:

```
#meu_canvas {
   width: 200px;
   height: 200px;
}
```

Contexto gráfico

Para desenhar no Canvas, é preciso executar um script após ele ter sido carregado. Neste script, obteremos o *contexto gráfico*, que é o objeto que realiza de fato as tarefas de desenho no Canvas.

Uma maneira é criar uma tag `<script>` após a tag `<canvas>`:

```
<!DOCTYPE html>
<html>

<head>
   <title>Processando o Canvas após a tag</title>
</head>

<body>
   <canvas id="meu_canvas" width="200" height="200"></canvas>
   <script>
      // Aqui obteremos o contexto gráfico e trabalharemos com o
      // Canvas
   </script>
</body>

</html>
```

Também é muito comum desenharmos em eventos que ocorrem após a página ter sido carregada. Isto é útil caso queiramos colocar os scripts na seção `<head>` do documento HTML:

```
<!DOCTYPE html>
<html>

<head>
   <title>Processando o Canvas na seção HEAD</title>
   <script>
      window.onload = function() {
         // Aqui trabalharemos com o Canvas
      }
   </script>
</head>

<body>
   <canvas id="meu_canvas" width="200" height="200"></canvas>
</body>

</html>
```

No código, nós referenciamos o Canvas e obtemos o contexto gráfico. O Canvas é referenciado como qualquer elemento em uma página; o contexto é obtido pelo método `getContext` do Canvas. Como parâmetro, passamos uma string identificando o contexto desejado. Neste livro, usaremos o contexto `2d` ("d" em minúsculo!):

```
<canvas id="meu_canvas" width="200" height="200"></canvas>
<script>
   // Referenciando o Canvas
   var canvas = document.getElementById('meu_canvas');

   // Obtendo o contexto gráfico
   var context = canvas.getContext('2d');
</script>
```

O sistema de coordenadas do Canvas

Para posicionarmos os desenhos no Canvas, pensamos nele como um enorme conjunto de pontos. Cada ponto possui uma posição horizontal (x) e uma vertical (y).

O ponto (0, 0) (lê-se: zero em `x` e zero em `y`) corresponde ao canto superior esquerdo do Canvas:

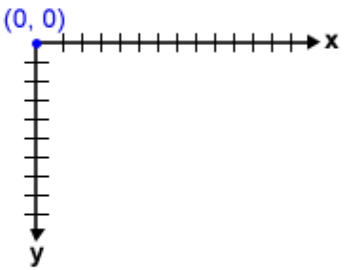


Figura 1.2: Sistema de coordenadas do Canvas

1.2 Começando a desenhar

Métodos *fillRect* e *strokeRect*

Uma vez obtido o contexto gráfico, podemos configurar várias propriedades e chamar nele os métodos de desenho. Por exemplo, para desenhar retângulos, podemos usar os métodos:

- `fillRect(x, y, largura, altura)`: pinta completamente uma área retangular;
- `strokeRect(x, y, largura, altura)`: desenha um contorno do retângulo.

Os valores `x` e `y` corresponderão à posição do canto superior esquerdo do retângulo. A partir daí, o retângulo vai para a direita (largura) e para baixo (altura).

Veja um exemplo de código para desenhar um retângulo no Canvas, nosso primeiro exemplo prático completo:

```
<!-- arquivo: retangulos-1.html -->
<!-- este código vai dentro do body -->
<canvas id="meu_canvas" width="200" height="200"></canvas>
<script>
   // Canvas e contexto
   var canvas = document.getElementById('meu_canvas');
   var context = canvas.getContext('2d');

   // Desenhar um retângulo
   context.fillRect(50, 50, 100, 100);
</script>
```

Não é simples de fazer? O resultado será um simples retângulo preto. Seu canto superior esquerdo localiza-se no ponto (50, 50), e ele possui 100 pixels de largura por 100 pixels de altura:

Figura 1.3: Retângulo desenhado com *fillRect*

Se trocarmos `fillRect` por `strokeRect`, veremos apenas o contorno:

Figura 1.4: Retângulo desenhado com *strokeRect*

Propriedades *fillStyle*, *strokeStyle* e *lineWidth*

Podemos configurar algumas propriedades do contexto, de forma a escolher as cores e espessuras:

- *fillStyle:* cor do preenchimento
- *strokeStyle:* cor da linha
- *lineWidth:* espessura da linha em pixels

```
<!-- arquivo: retangulos-2.html -->
<canvas id="meu_canvas" width="200" height="200"></canvas>
<script>
   // Canvas e contexto
   var canvas = document.getElementById('meu_canvas');
   var context = canvas.getContext('2d');

   // Preenchimento vermelho
   context.fillStyle = 'red';
   context.fillRect(50, 50, 100, 100);

   // Contorno azul, com espessura de 3px
   context.lineWidth = 3;
   context.strokeStyle = 'blue';
   context.strokeRect(50, 50, 100, 100);
</script>
```

Figura 1.5: Configurando as propriedades do preenchimento (*fillStyle*) e contorno (*strokeStyle*)

Paths (caminhos)

Desenhos mais complexos podem ser desenhados como *paths* (caminhos). Um *path* é um conjunto de comandos de desenho que ficam registrados na memória, aguardando os métodos *fill* (preencher) ou *stroke* (contornar) serem chamados.

Porém, antes de tudo, devemos chamar o método *beginPath* (iniciar caminho) para apagar os traçados feitos previamente. Se não fizermos isso, eles ficarão na memória e serão desenhados novamente junto com o próximo path:

```
// Exemplo teórico
// Passo a passo para criar um path:

// Iniciar novo path (apagando desenhos anteriores)
context.beginPath();

// Aqui faço meu desenho
// ...

// Preencher toda a área desenhada
context.fill();

// Contornar a área desenhada
context.stroke();
```

Por exemplo, podemos desenhar uma estrela usando os seguintes comandos:

- `moveTo(x, y)`: posiciona a caneta virtual em um determinado ponto;
- `lineTo(x, y)`: traça uma linha do ponto atual até o ponto indicado.

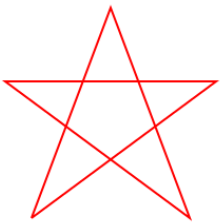


Figura 1.6: Estrela criada usando *moveTo* e *lineTo*

Esses comandos não desenham as linhas imediatamente, apenas armazenam as informações no path. Devemos chamar o método `stroke` para desenhá-las de fato no Canvas:

```
<!-- arquivo: caminhos.html -->
<canvas id="meu_canvas" width="300" height="300"></canvas>
<script>
   // Canvas e contexto
   var canvas = document.getElementById('meu_canvas');
   var context = canvas.getContext('2d');

   // Iniciar o caminho (apaga desenhos anteriores)
   context.beginPath();

   // Desenhar uma estrela
   context.moveTo(75, 250);  // Ponto inicial
   context.lineTo(150, 50);
   context.lineTo(225, 250);
   context.lineTo(50, 120);
   context.lineTo(250, 120);
   context.lineTo(75, 250);

   // Configurar a linha
   context.lineWidth = 2;
   context.strokeStyle = 'red';

   // Traçar as linhas do caminho
   context.stroke();
```

```
</script>
```

Circunferências e arcos

São criados com o mesmo método, `arc`. Um arco é uma parte de uma circunferência, e serve para criar linhas curvas. Uma circunferência, para o Canvas, é nada mais que um arco de 360 graus.

O método `arc` recebe os seguintes parâmetros:

- *x, y:* as coordenadas do ponto central da circunferência;
- *raio* da circunferência em pixels;
- *ângulo inicial* em radianos;
- *ângulo final* em radianos;
- *sentido:* aqui você pode passar `false` (sentido anti-horário) ou `true` (sentido horário). Este parâmetro é opcional; se omitido, o desenho é feito no sentido anti-horário.

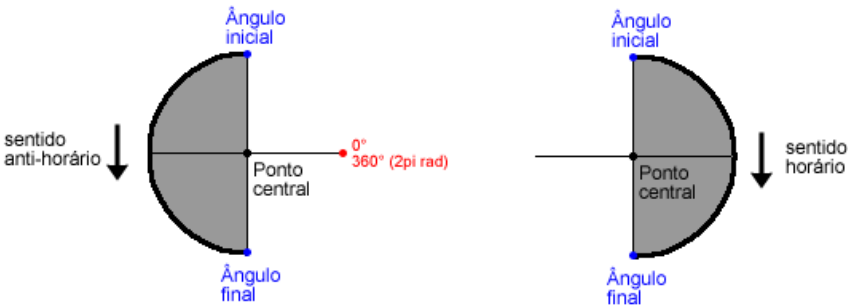


Figura 1.7: Localização do ponto central, ângulos inicial e final e o ponto zero

Os ângulos podem estar em qualquer posição na circunferência e têm de ser informados em *radianos*, não em graus. Lembra-se das aulas de trigonometria? Eu também não, mas sabemos que *pi* radianos é meia volta (180 graus), portanto *2 pi* vale uma volta completa (360 graus). Para

nossa sorte, o JavaScript possui a constante `Math.PI`, que nos dá o valor `3.141592653589793`!

Convertendo de graus para radianos

A fórmula é muito simples:

```
var graus = ... ;
var radianos = graus * Math.PI / 180;
```

Importante: o método `arc` é usado em um path, ou seja, não desenha o arco ou círculo na hora.

O primeiro arco da figura 1.7 pode então ser desenhado com o comando:

```
// Exemplo teórico
context.arc(
   50,               // X (centro)
   50,               // Y (centro)
   40,               // Raio
   90*Math.PI/180,   // Início: 90 graus
   270*Math.PI/180,  // Término: 270 graus
   false             // Sentido anti-horário
);
```

Para fazer o outro arco, basta escrevermos `true` no último parâmetro.

Vamos experimentar na prática! Primeiro, vamos inicializar as variáveis e configurar o contexto com a aparência desejada:

```
<!-- arquivo: arcos.html -->
<canvas id="meu_canvas" width="300" height="300"></canvas>
<script>
   // Canvas e contexto
   var canvas = document.getElementById('meu_canvas');
   var context = canvas.getContext('2d');

   // Cores e espessura da linha
   context.fillStyle = 'gray';
```

```
    context.strokeStyle = 'black';
    context.lineWidth = 2;

    // continua...
</script>
```

Para fazer um arco, devemos iniciar um path, chamar o método `arc` e depois fazer `fill` e `stroke` (ou somente um destes dois, se quisermos somente preenchimento ou somente contorno):

```
// Primeiro arco:
// Novo path
context.beginPath();
// Desenha
context.arc(50, 50, 40, 90*Math.PI/180, 270*Math.PI/180,
    false);
// Preenchimento
context.fill();
// Contorno
context.stroke();

// continua...
```

Se quisermos fazer outro arco, temos que iniciar mais um path; do contrário, o desenho formado ficará estranho (figura 1.8). Repare que a posição `x` foi deslocada para desenharmos à direita do anterior:

```
// Segundo arco
context.beginPath();
context.arc(150, 50, 40, 90*Math.PI/180, 270*Math.PI/180,
    true);
context.fill();
context.stroke();

// continua ...
```

Figura 1.8: Dois arcos desenhados juntos no mesmo path: um é a continuação do outro!

Faça também uma circunferência completa. Não precisa se preocupar em converter 360 graus para radianos, basta fazer o arco de zero até *2 pi*. Vamos omitir o último parâmetro, pois tanto faz fazermos no sentido horário ou anti-horário (já que vamos começar e finalizar no mesmo ponto):

```
// Circunferência completa
context.beginPath();
context.arc(250, 50, 40, 0, 2*Math.PI);
context.fill();
context.stroke();
```

Pronto! Temos os seguintes desenhos na página:

Figura 1.9: Arco em sentido anti-horário, arco em sentido horário e circunferência completa, com preenchimento e contorno

Desenhando imagens

Se você já está ficando preocupado, pensando se todos os gráficos em um jogo são criados via programação, eu tenho uma boa notícia para você: não

são. A grande maioria vem de imagens já prontas, elaboradas em programas gráficos.

Para desenhar imagens pré-elaboradas em um Canvas, primeiro temos de carregar o arquivo de imagem. O objeto `Image` do JavaScript é equivalente a um elemento `<img>` na página, porém somente em memória. Após criá-lo, apontamos seu atributo `src` para o arquivo desejado:

```
// Exemplo teórico

// Carregando uma imagem programaticamente
var imagem = new Image();
imagem.src = 'minha-imagem.png';  // gif, jpg...

// Obtendo de uma imagem na página
var imagem = document.getElementById('id_da_tag_img');
```

Convém aguardar a imagem ser carregada antes de desenhá-la. O objeto `Image` possui o evento `onload`, que será disparado automaticamente pelo browser quando o carregamento estiver completo:

```
// Exemplo teórico
imagem.onload = function() {
   // Aqui trabalhamos com a imagem
}
```

Estando carregada, a imagem pode ser desenhada através do método `drawImage` do `context`. Este método pode ser chamado de duas formas:

- `drawImage(imagem, x, y, largura, altura)`: desenha a imagem inteira, na posição e tamanho especificados;
- `drawImage(imagem, xOrigem, yOrigem, larguraOrigem, alturaOrigem, xDestino, yDestino, larguraDestino, alturaDestino)`: desenha parte da imagem.

Vamos experimentar a primeira forma. No pacote de download, na pasta deste capítulo (`01`), está presente uma subpasta de nome `img`, contendo o arquivo `ovni.png`. Este será posteriormente o inimigo que a nave irá

enfrentar em nosso jogo. A imagem possui 64 pixels de largura por 32 de altura (64x32). Vamos criar uma página com um Canvas e carregá-la:

```
<!-- arquivo: imagens-1.html -->
<canvas id="meu_canvas" width="500" height="100"></canvas>
<script>
   // Canvas e contexto
   var canvas = document.getElementById('meu_canvas');
   var context = canvas.getContext('2d');

   // Carregar a imagem
   var imagem = new Image();
   imagem.src = 'img/ovni.png';
   imagem.onload = function() {
      // Aqui usaremos drawImage
   }
</script>
```

No evento `onload`, fazemos um loop para desenhar cinco OVNIs, um ao lado do outro. A variável `x` indica a posição de cada desenho:

```
imagem.onload = function() {
   // Começar na posição 20
   var x = 20;

   // Desenhar as imagens
   for (var i = 1; i <= 5; i++) {
      context.drawImage(imagem, x, 20, 64, 32);
      x += 70;
   }
}
```

Figura 1.10: Desenhando imagens com *drawImage*

Se quiséssemos desenhar as imagens ampliadas ou reduzidas, bastaria modificar a largura e a altura:

```
// Reduzindo para metade do tamanho
context.drawImage(imagem, x, 20, 32, 16);

// Ampliando para o dobro do tamanho
context.drawImage(imagem, x, 20, 128, 64);
```

Vamos experimentar a segunda forma, a que recebe oito valores! Na pasta `img`, temos o arquivo `explosao.png`, que também será usado no jogo. Esta imagem contém uma sequência de animação (*spritesheet*), da qual desenhamos uma parte por vez:

Figura 1.11: Explosão usada no jogo do livro

A técnica de *clipping* consite em selecionar uma área da imagem original para ser desenhada:

Figura 1.12: Selecionando uma área da imagem para recorte (*clipping*)

Os quatro primeiros valores passados ao `drawImage` indicam o retângulo da área enquadrada; os outros quatro representam a posição e o tamanho do desenho no Canvas:

```
<!-- arquivo: imagens-2.html -->
<canvas id="meu_canvas" width="300" height="300"></canvas>
<script>
   // Canvas e contexto
   var canvas = document.getElementById('meu_canvas');
   var context = canvas.getContext('2d');

   // Carregar a imagem
   var imagem = new Image();
   imagem.src = 'img/explosao.png';
   imagem.onload = function() {
      context.drawImage(
         imagem,
         80, 10, 60, 65,  // Área de recorte (clipping)
         20, 20, 60, 65   // Desenho no Canvas
      );
   }
</script>
```

Figura 1.13: Resultado do clipping no Canvas

No capítulo 4, faremos uso intensivo do clipping para criar animações.

Métodos *save* e *restore*

Um recurso do `context` que considero de extrema importância é poder facilmente guardar configurações e retornar a elas mais tarde. É uma excelente prática que cada função ou método guarde as configurações atuais e as restaure antes de retornar. Desta forma, um não afeta o trabalho do outro!

Mas já imaginou ter que salvar cada propriedade em uma variável?

```
// Exemplo teórico
// Não faça isto!
function desenharQualquerCoisa() {
   // Guardo as configurações
   var fillAnterior = context.fillStyle;  // strokeStyle, etc.

   // Modifico conforme preciso
   context.fillStyle = 'pink';

   // Desenho
   // ...

   // Devolvo as configurações anteriores
   context.fillStyle = fillAnterior;
}
```

Para nossa sorte, o `context` possui o método `save`. Este método empilha configurações, fazendo uma cópia de *todas* as configurações atuais para o próximo nível. Pense em cada nível como um "andar" nessa pilha. Nós sempre trabalhamos no nível mais alto, de modo que configurações que ficaram em níveis inferiores permanecem inalteradas:

Figura 1.14: Método *save* empilhando configurações

Quando queremos retornar às configurações anteriores, chamamos o método `restore`. Ele volta para o nível imediatamente abaixo:

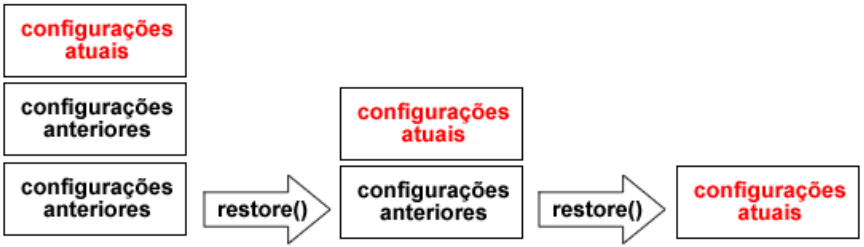


Figura 1.15: Voltando para configurações anteriores com *restore*

Como exemplo prático, começamos desenhando um quadrado verde. Em seguida, subimos na pilha com `save` e fazemos um quadrado roxo. Depois que retornamos com `restore`, nossa cor atual passa a ser o verde novamente:

```
<!-- arquivo: save-e-restore.html -->
<canvas id="meu_canvas" width="200" height="200"></canvas>
<script>
   // Canvas e contexto
   var canvas = document.getElementById('meu_canvas');
   var context = canvas.getContext('2d');

   // Um pequeno quadrado verde
   context.fillStyle = 'green';
   context.fillRect(50, 50, 25, 25);

   // Salvamos a configuração e subimos na pilha
   context.save();

   // Agora um quadrado roxo
   context.fillStyle = 'purple';
   context.fillRect(100, 50, 25, 25);

   // Voltamos para o nível anterior na pilha
   context.restore();

   // Voltou para o verde!
```

```
    context.fillRect(150, 50, 25, 25);
</script>
```

Figura 1.16: Salvamos a cor verde na pilha (*save*), mudamos para roxo e voltamos para o verde (*restore*)

Referências sobre o Canvas

Neste tópico, não quis nem de longe esgotar os recursos do Canvas, mas dar uma base para iniciarmos o desenvolvimento de jogos. Indico-lhe algumas referências para você aprimorar seus conhecimentos e ter ideias, muitas ideias, para implementar em seus jogos:

- *W3Schools*[3] (http://www.w3schools.com/tags/ref_canvas.asp) : recomendo todos os tutoriais deles, por serem simples e por você poder rodar os exemplos na hora;
- *Core HTML5 Canvas*[1] (http://corehtml5canvas.com) : excelente livro de David Geary, bem aprofundado, foi a minha principal fonte para aprender sobre o Canvas. O autor discute não somente os recursos da ferramenta, mas técnicas e algoritmos extremamente avançados e úteis para jogos;
- *HTML5 2D game development: Introducing Snail Bait*[2] (http://goo.gl/Ojvi9T - URL encurtada): do mesmo autor do Core HTML5 Canvas, este é o primeiro de uma série de artigos na qual somos guiados na criação de um jogo de plataforma.

1.3 Animações com *requestAnimationFrame*

Tradicionalmente, quando queremos executar tarefas periódicas em páginas web, usamos os métodos `window.setTimeout` ou `window.setInterval`, passando a função desejada e o intervalo em milissegundos. Os resultados são satisfatórios para tarefas simples, no entanto, o controle do tempo pelo browser não é totalmente preciso. É preciso lembrar que os sistemas operacionais modernos são multitarefa, e mesmo os browsers podem ter várias guias abertas. Não é possível garantir que a CPU esteja sempre disponível no momento exato desejado, portanto o intervalo informado sempre será aproximado.

A solução para jogos e outras aplicações que requerem animações mais precisas é trabalhar com ciclos mais curtos, aproveitando cada momento em que a CPU está disponível para nosso aplicativo, e nesse momento fazer o cálculo do tempo. Para isso, a especificação do HTML5 traz o método `window.requestAnimationFrame`, que delega para o browser a tarefa de executar sua animação o mais rápido possível, assim que os recursos do sistema estiverem disponíveis. Para este método, passamos como parâmetro a função que fará os desenhos no Canvas. Temos que chamá-lo de forma cíclica, uma vez após a outra, da mesma forma que faríamos caso usássemos o conhecido `setTimeout`:

```
// Exemplo teórico

// Solicito a execução de uma função
// Não é preciso qualificar o "window" :D
requestAnimationFrame(minhaFuncao);

// Função de animação
function minhaFuncao() {
   // Faço um desenho qualquer
   // ...

   // Solicito o próximo ciclo
   requestAnimationFrame(minhaFuncao);
}
```

Para demonstrar na prática, façamos uma bolinha se deslocando pela tela.

Obtemos as referências do Canvas e do contexto gráfico, definimos a posição inicial da bola e seu raio e, em seguida, mandamos uma animação iniciar, chamando a função `mexerBola`:

```
<!-- arquivo: requestanimationframe.html -->
<canvas id="meu_canvas" width="300" height="300"></canvas>
<script>
   // Canvas e contexto
   canvas = document.getElementById('meu_canvas');
   context = canvas.getContext('2d');

   // Dados da bola
   var x = 20;
   var y = 100;
   var raio = 5;

   // Iniciar a animação
   requestAnimationFrame(mexerBola);

   // Função de animação
   function mexerBola() {
      // Aqui uma bolinha se deslocará
   }
</script>
```

Nessa função, primeiro limpamos o Canvas com o método `clearRect` do contexto, que serve para limpar uma área (equivale a desenhar um retângulo branco). Fazemos isso para apagar rastros anteriores da bolinha. Em seguida, nós a desenhamos em sua posição atual, alteramos sua posição x e solicitamos a execução do próximo ciclo de animação:

```
function mexerBola() {
   // Limpar o Canvas
   context.clearRect(0, 0, canvas.width, canvas.height);

   // Desenhar a bola
   context.beginPath();
   context.arc(x, y, raio, 0, Math.PI*2);
   context.fill();
```

```
    // Deslocar 20 pixels para a direita
    x += 20;

    // Chamar o próximo ciclo da animação
    requestAnimationFrame(mexerBola);
}
```

Faça experiências! Você pode:

- *mexer para a direita:* some um valor a *x* (veja a figura 1.2);
- *mexer para a esquerda:* subtraia um valor de *x*;
- *mexer para baixo:* some um valor a *y*;
- *mexer para cima:* subtraia um valor de *y*;
- *mexer na diagonal:* altere as posições tanto de *x* quanto de *y*. Você pode somar ou subtrair, usar valores diferentes para cada um etc.

Controlando o tempo da animação

Repare que a bolinha anda bem depressa. Isto ocorre porque o `requestAnimationFrame` trabalha com ciclos curtos, aproveitando o primeiro momento em que a CPU e o browser puderem executar o processamento da função de animação.

Podemos ler o relógio do computador em cada ciclo para controlar o movimento da bolinha. Sabemos que o JavaScript possui o objeto `Date`, que obtém a data e a hora atuais, e que esse objeto possui o método `getTime()`, que devolve esse instante exato em milissegundos. Para saber quanto tempo demorou entre um ciclo e outro (lembre-se de que esse tempo é sempre variável), bastaria tirar a diferença entre o instante atual e o anterior. Sabendo o intervalo decorrido, é possível calcular quanto a bola deve se deslocar nesse tempo:

```
// Exemplo teórico

// Obter o instante atual
```

```
var agora = new Date().getTime();

// O instante anterior tem de ter sido preservado anteriormente
var anterior = agora;  // Foi feito antes

// Tempo decorrido = diferença
var decorrido = agora - anterior;

// Deslocamento da bolinha
var velocidade = 20;  // Em pixels por segundo
x += velocidade * decorrido / 1000;
```

Vamos colocar isto em prática. Primeiro, guarde o momento inicial antes de chamar a função de animação pela primeira vez:

```
// Momento inicial
var anterior = new Date().getTime();

// Iniciar a animação
requestAnimationFrame(mexerBola);

// Função de animação
function mexerBola() {
   // ...
}
```

No início da função, obtenha o instante atual e calcule o tempo decorrido entre eles:

```
function mexerBola() {
   // Momento atual
   var agora = new Date().getTime();

   // Diferença
   var decorrido = agora - anterior;

   // ...
}
```

Ao final da função, antes de chamar o próximo ciclo, guarde o momento do ciclo anterior:

```
function mexerBola() {
   // ...

   // Guardamos o instante para o próximo ciclo
   anterior = agora;
   requestAnimationFrame(mexerBola);
}
```

Agora podemos calcular o desclocamento da bola! Apague a linha que aumenta *x* em 20 pixels:

```
// Apague esta linha
// Deslocar 20 pixels para a direita
x += 20;
```

No lugar, calcule o deslocamento a partir da velocidade de 20 pixels por segundo (ou outra que preferir):

```
// Deslocar 20 pixels por segundo
var velocidade = 20;
x += velocidade * decorrido / 1000;
```

A bola agora move-se mais lentamente, pois agora estamos trabalhando com pixels por segundo (antes eram 20 pixels por ciclo, e tínhamos inúmeros ciclos em um segundo). Para fazê-la mover-se mais rápido, basta aumentar o valor.

1.4 Orientação a objetos com JavaScript

Conceitos básicos

Não é intenção deste livro ensinar a teoria da orientação a objetos (O.O.), mas vou dar-lhe uma base para podermos usá-la com JavaScript. Esta não é uma linguagem puramente O.O., no entanto, o jogo que criaremos, está projetado dessa forma. Diferenciamos aqui *projeto orientado a objetos* (aplicado em nosso jogo) de *linguagem orientada a objetos*, que dá suporte total ao paradigma (não é o caso do JavaScript).

Imagine um jogo de carros de corrida. Todos os carros são padronizados nas suas características, embora elas possam variar (dentro desse padrão). Por exemplo, todos os carros possuem cor, velocidade máxima e velocidade atual. Contudo, em um dado momento, cada carro possui *valores* diferentes para estas características. O carro do jogador é vermelho, está a 150km/h e pode alcançar até 200km/h. Seu rival é azul, está a 170km/h e pode alcançar até 220km/h. As características que definem um carro no jogo são as mesmas, mas cada uma está preenchida com valores diferentes para cada carro.

Esse conjunto de características comuns a todo carro é o que chamamos de *classe*. Quando falamos *carro*, referindo-nos a *carros em geral*, estamos falando de uma classe. A classe é o molde para a criação de objetos.

Cada carro é uma *instância* da classe. A instância é o objeto em si, um objeto que pertence a uma classe. Quando falo do *meu carro*, estou falando de uma instância específica da classe *carro*. O *carro do vizinho* é outra instância.

Cor, velocidade atual e velocidade máxima são os *atributos* da classe, ou seja, todas as instâncias os têm, mas cada uma possui valores próprios para esses atributos.

Cada carro executa as tarefas de acelerar, frear e virar. Estas tarefas são os *métodos*. Em programas orientados a objeto, é nos métodos que escrevemos os códigos que realizam as tarefas fundamentais do programa. Eles trabalham com os dados nos atributos e podem alterar o estado do objeto.

Resumo de conceitos básicos de O.O.

- *Classe:* um tipo de objetos determinado, composto por atributos e métodos definidos;
- *Instância:* cada objeto pertencente a uma determinada classe;
- *Atributo:* cada propriedade dos objetos listada em uma classe;
- *Método:* cada tarefa que um objeto de uma classe pode executar.

Funções construtoras

Em linguagens O.O., como Java e C#, definimos classes explicitamente (usando a palavra-chave `class`). Em JavaScript não há classes, mas podemos usar *funções construtoras* para criar os objetos seguindo as classes que projetamos.

Uma função construtora é uma função normal do JavaScript, porém usada com o propósito de criar objetos. No exemplo a seguir, a palavra-chave `this` recebe os atributos de um carro. A palavra `cor`, sozinha, se refere ao parâmetro recebido pela função, mas a construção `this.cor` cria um atributo `cor` no objeto:

```
<!-- arquivo: orientacao-objetos.html -->
<body>
   <script>
      function Carro(cor, velocMaxima) {
         this.cor = cor;
         this.velocMaxima = velocMaxima;
         this.velocAtual = 0;
      }

      // continua...
   </script>
</body>
```

Nós não podemos criar um carro simplesmente chamando:

```
// Exemplo teórico
// Não faça isto:
Carro('verde', 300);
```

Quando chamamos uma função sem qualificar um objeto, o JavaScript entende que estamos usando o objeto `window`. Assim, a palavra `this` no corpo da função criará ou modificará os atributos neste objeto.

Para criar novos objetos temos de usar a palavra-chave `new`. Após a função construtora, crie as variáveis `meuCarro` e `oponente` para referenciar os novos objetos:

```
// Instanciando objetos da classe Carro
var meuCarro = new Carro('vermelho', 250);
var oponente = new Carro('azul', 300);
```

A expressão `new Carro('vermelho', 250)` cria um novo objeto e executa nele a função construtora. Agora a palavra `this` se referirá a esse objeto!

Já os métodos, em JavaScript, são apenas atributos que referenciam funções. Podemos, por exemplo, criar um método `acelerar` que aumenta em 10 unidades a velocidade do carro:

```
function Carro(cor, velocMaxima) {
   // Criando os atributos
   this.cor = cor;
   this.velocMaxima = velocMaxima;
   this.velocAtual = 0;

   // Criando um método
   this.acelerar = function() {
      this.velocAtual += 10;
   }
}

// ...
```

Chamando o método `acelerar` através da variável `meuCarro` criada anteriormente, esse método vai alterar o atributo `velocAtual` dessa instância, e não da instância `oponente`:

```
// Chamando um método somente em um objeto
meuCarro.acelerar();
```

Por último, mande exibir as velocidades dos dois carros criados. O primeiro acelerou, e o outro não!

```
// Verificando as velocidades
document.write(meuCarro.cor + ': ' + meuCarro.velocAtual);
document.write('<br>');
document.write(oponente.cor + ': ' + oponente.velocAtual);
```

A saída desse código é:

```
vermelho: 10
azul: 0
```

Sintaxe de ponto e de colchetes

Para referenciarmos atributos de um objeto em JavaScript, o mais usual é usarmos a *sintaxe de ponto*, ou seja, usar um ponto entre o nome do objeto e do atributo:

```
meuObjeto.atributo
```

Também é possível usar a *sintaxe de colchetes*, na qual podemos usar strings para representar os atributos. O objeto torna-se uma espécie de array associativo, sendo que cada elemento (atributo) é associado a uma string específica, e não a um índice de posição:

```
meuObjeto['atributo']
```

Objetos sem função construtora

O JavaScript oferece a sintaxe de chaves (`{` e `}`) para criar objetos:

```
var meuObjeto = {
   atributo1: valor,
   atributo2: valor,
   metodo1: function() {

   },
   metodo2: function() {

   }
}
```

Por exemplo, poderíamos criar um carro da seguinte forma:

```
var meuCarro = {
   cor: 'azul',
   velocidade: 0,
   acelerar: function() {
      this.velocidade += 10;
   }
}
```

Nós usaremos essa forma algumas vezes no desenvolvimento do jogo. Como, porém, seu projeto é orientado a objetos, pensando em *classes*, com atributos e métodos definidos, daremos maior preferência às funções construtoras.

O *prototype* (protótipo)

Declarar métodos como funções anônimas dentro das construtoras tem um preço: maior consumo de memória. Cada vez que executamos o construtor, uma cópia da função anônima é criada na memória e cada instância terá sua cópia não somente do atributo, mas também da função:

```
function Carro(cor, velocMaxima) {
   // ...

   // Criando método com função anônima
   this.acelerar = function() {
      this.velocAtual += 10;
   }
}
```

Pensando nisso, o JavaScript traz o recurso do *prototype* (protótipo), que é um objeto associado a uma função construtora. Colocando os métodos neste objeto, todas as instâncias usarão as mesmas cópias de cada método.

Vamos nos habituar a colocar a função construtora e o *prototype* de uma classe em um arquivo separado, com extensão `.js`. A palavra-chave `prototype` funciona como se fosse um atributo da função construtora:

```
// arquivo: carro.js
```

```
// Função construtora
function Carro(cor, velocMaxima) {
   this.cor = cor;
   this.velocMaxima = velocMaxima;
   this.velocAtual = 0;
}

// Prototype com os métodos
Carro.prototype = {
   acelerar: function() {
      this.velocAtual += 10;
   }
}
```

Escrevemos um pouco mais de código, porém até ganhamos um pouco mais de legibilidade e organização: conseguimos "enxugar" a função construtora, e podemos ter o funcionamento completo de um carro em seu arquivo específico — para *isto* servem as classes! Para utilizar essa classe em uma página HTML, basta usar a tag `<script>` com o atributo `src`, referenciando o arquivo `carro.js`:

```
<!-- arquivo prototype.html -->
<!DOCTYPE html>
<html>

<head>
   <title>0 prototype</title>
   <script src="carro.js"></script>
</head>

<body>
   <script>
      var meuCarro = new Carro('vermelho', 250);
      meuCarro.acelerar();
      document.write(meuCarro.cor + ': ' + meuCarro.velocAtual);
   </script>
</body>

</html>
```

É isso! Com estes conceitos básicos bem fixados, vamos começar a aprender conceitos específicos de jogos. No próximo capítulo, vamos aprender uma das bases de um jogo, que consiste no loop de animação e nos *sprites*.

Capítulo 2

O loop de animação

2.1 Introdução e sprites

Chegou a hora de avançarmos um passo além de apenas desenhar no Canvas e movimentar alguns objetos. Vamos desenvolver pequenos aplicativos e criar pequenas peças que, em algum momento, irão se juntar e tomar a forma de um jogo!

Nossa abordagem será a seguinte:

- *Primeiro, sempre codificamos o "aplicativo principal", que realiza um teste nas classes a serem criadas.*

Isso é importante para chegarmos a códigos claros e bem estruturados. Ao desenvolver, sempre procuramos focar em um aspecto por momento e abstrair ("deixar para lá") os outros. O aplicativo de teste determina o que esperamos de nossas classes.

- *Em seguida, codificamos as classes e tentamos fazê-las responder como queremos.*

Nessa etapa, sempre surgirão modificações na forma como planejamos inicialmente, o que vemos como algo natural. *Refatorações* (mudanças na estrutura do código sem alterar seu comportamento) serão rotina para nós.

> Mais do que ensinar como se faz, pretendo apresentá-lo a uma forma de trabalhar que deixará o resultado muito mais claro e organizado no final. Você me agradecerá, garanto-lhe.

Temos de começar por algum lugar, então escolhi o que é mais comum a todo jogo: o loop de animação. Criaremos uma classe responsável por controlar esse loop.

Antes de começar, vamos ter em mente que problemas ela resolverá. "Mas fazer um loop de animação não é só chamar o método repetidamente?", você dirá. Depende do que você quer fazer. Em um jogo:

- a animação tem que ser parada e reiniciada em diversos momentos;
- vários objetos serão animados em cada ciclo;
- objetos podem ser incluídos e excluídos da animação.

Conceito de sprites

Aos objetos trabalhados dentro de um loop de animação, damos o nome de **sprites**. Pense em um sprite como sendo uma imagem com fundo transparente, que pode ser animada. Dessa forma, eles podem ser inseridos sobre o fundo principal do cenário.

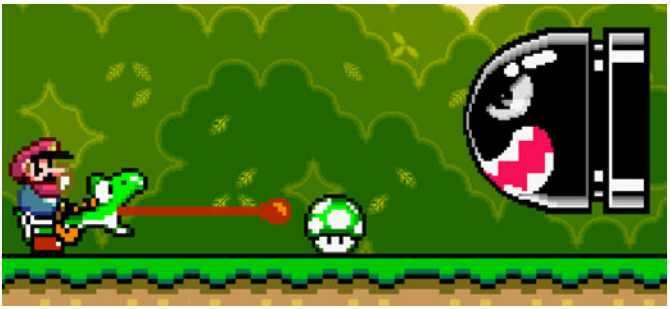


Figura 2.1: Sprites

Sprites são criados e destruídos a todo momento. Por "destruídos", entenda não apenas morto, aniquilado, atingido, mas destruídos na *memória do computador*. Por exemplo, um inimigo ou item que saiu da tela pode ser destruído e recriado caso o herói retorne àquele ponto. Em JavaScript, não fazemos destruição de objetos (em outras linguagens isso é possível), apenas anulamos as variáveis que se referem a ele e o deixamos livres para o *garbage collector* apagá-lo da memória automaticamente.

Chega de conceitos! É hora de pôr a mão na massa.

2.2 Teste para a classe Animacao

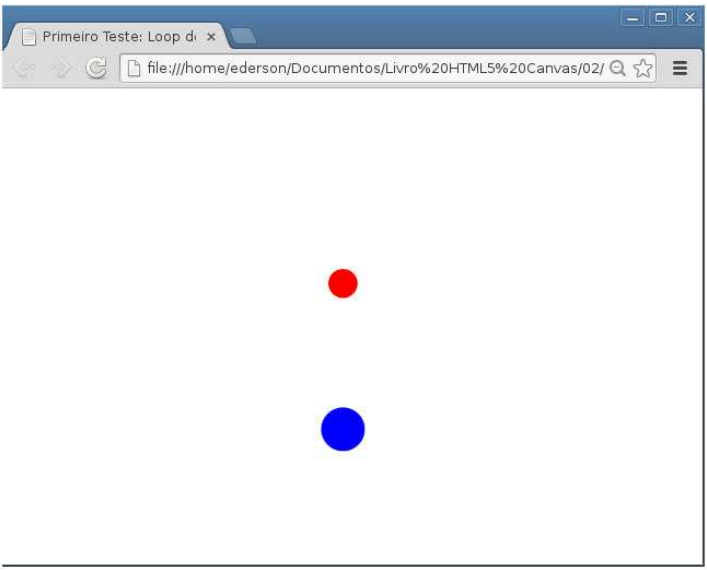


Figura 2.2: Animação controlada por classe JavaScript

Como prometemos, faremos um pequeno aplicativo de teste no qual definimos como queremos "mandar" nas classes que criaremos. Como estamos desenvolvendo para a web, nosso aplicativo será uma pequena página HTML com alguns códigos JavaScript.

A página referencia dois arquivos JavaScript: `animacao.js`, que conterá a classe que controlará o loop de animação, e `bola.js`, onde definiremos um sprite (uma bola que quica na tela).

```
<!-- arquivo: animacao-teste.html -->
<!DOCTYPE html>
<html>

<head>
   <title>Primeiro Teste: Loop de Animação com Sprites</title>
   <script src="animacao.js"></script>
   <script src="bola.js"></script>
```

```
</head>

<body>
   <canvas id="canvas_animacao" width="500" height="500">
   </canvas>
   <script>
      // Aqui vai a inicialização do aplicativo
   </script>
</body>

</html>
```

Após o Canvas, a página executará um script de inicialização, cujo código vem a seguir. Como aprendemos no capítulo anterior, tópico 1.1, primeiro temos de recuperar o Canvas e seu contexto *2d*. Depois, criamos os primeiros sprites *b1* e *b2* para as bolas, indicando suas coordenadas na tela, e outras características como velocidade, cor e raio.

```
// Referências para o Canvas
var canvas = document.getElementById('canvas_animacao');
var context = canvas.getContext('2d');

// Criando alguns sprites
var b1 = new Bola(context);
b1.x = 100;
b1.y = 200;
b1.velocidadeX = 20;
b1.velocidadeY = -10;
b1.cor = 'red';
b1.raio = 20;

var b2 = new Bola(context);
b2.x = 200;
b2.y = 100;
b2.velocidadeX = -10;
b2.velocidadeY = 20;
b2.cor = 'blue';
b2.raio = 30;
```

Em seguida, vamos criar a nossa animação hipotética, incluir nela os sprites e ligá-la, fazendo com que se inicie imediatamente:

```
// Criando o loop de animação
var animacao = new Animacao();
animacao.novoSprite(b1);
animacao.novoSprite(b2);

// "Ligar" a animação
animacao.ligar();
```

Claro que, ao tentar abrir esta página no navegador, ela ainda *não* funciona. Tecle `Ctrl+Shift+J` no Google Chrome, ou `Ctrl+Shift+K` no Firefox, e delicie-se com estas mensagens:

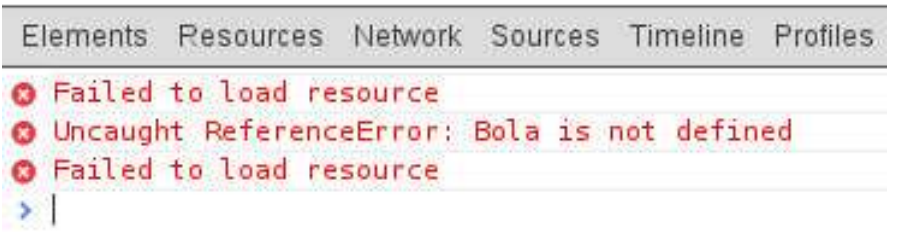


Figura 2.3: Erros no Console do Google Chrome na fase de escrita do teste

A razão é que não temos ainda os arquivos `animacao.js` e `bola.js`. Nós apenas definimos como queremos interagir com as classes que ainda não saíram do ovo.

E como queremos mandar nessas classes? Quanto à `Bola`, quero poder criar quantas desejar, configurar posição, velocidades horizontal e vertical, raio e cor. Quanto à `Animacao`, no código estou dizendo que quero poder incluir sprites à vontade e mandar iniciá-la no momento que eu achar oportuno.

2.3 Desenvolva a classe Animacao

O "grosso" do trabalho começa aqui. Já temos bem definido *o que* queremos da classe neste ponto, mas agora falta explicitar *como*. Por mais desafiador que pareça, vamos por etapas bem pequenas (*"baby steps"*), e logo teremos um resultado concreto.

Primeiro, faremos o esqueleto da classe, com sua função construtora e os métodos que imaginamos no teste:

```
// Arquivo: animacao.js
function Animacao() {

}
Animacao.prototype = {
   novoSprite: function(sprite) {

   },
   ligar: function() {

   }
}
```

Para incluir um sprite na animação, podemos iniciar um array vazio no construtor e incluir nele os sprites:

```
function Animacao() {
  this.sprites = [];
}
Animacao.prototype = {
   novoSprite: function(sprite) {
      this.sprites.push(sprite);
   },
   ligar: function() {

   }
}
```

Agora vamos criar o atributo `ligado`, que definirá se a animação pode ou não correr. Aproveitando o ensejo, já criaremos também o método `desligar`:

```
function Animacao() {
   this.sprites = [];
   this.ligado = false;
}

Animacao.prototype = {
   novoSprite: function(sprite) {
      this.sprites.push(sprite);
   },

   ligar: function() {
      this.ligado = true;
      this.proximoFrame();
   },
   // Não esquecer desta vírgula sempre que
   // for criar um novo método!

   desligar: function() {
      this.ligado = false;
   }
}
```

Em `ligar`, chamo um novo método, `proximoFrame`, que será o coração da classe `Animacao`. Este método verifica se pode continuar e, se puder, realiza um ciclo da animação e chama a si mesmo novamente (usando o `requestAnimationFrame` do HTML5).

```
proximoFrame: function() {
   // Posso continuar?
   if ( ! this.ligado ) return;

   // A cada ciclo, limpamos a tela ou desenhamos um fundo
   this.limparTela();

   // Atualizamos o estado dos sprites
   for (var i in this.sprites)
      this.sprites[i].atualizar();

   // Desenhamos os sprites
```

```
    for (var i in this.sprites)
        this.sprites[i].desenhar();

    // Chamamos o próximo ciclo
    var animacao = this;
    requestAnimationFrame(function() {
        animacao.proximoFrame();
    });
}
```

Aqui temos muitas coisas acontecendo. A cada ciclo de animação, devemos:

- *limpar a tela:* se não fizermos isso, a bola deixará um rastro (figura 2.4). É preciso apagar os desenhos do ciclo anterior para desenhar a bola em seu novo estado, na nova posição que vai atualizar e desenhar logo em seguida. O método `limparTela` será implementado em breve;
- *atualizar o estado dos sprites:* cada sprite será responsável por se posicionar e realizar seus próprios comportamentos. Deixamos a cargo da `Bola` a função de calcular sua trajetória;
- *desenhar os sprites:* somente depois que todos estiverem atualizados;
- *chamar o próximo ciclo* da animação.

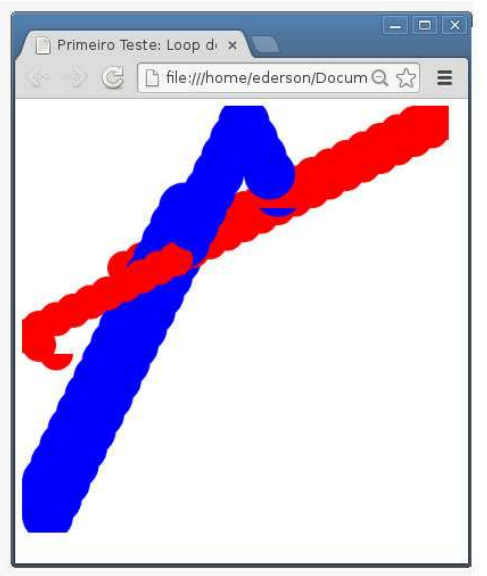


Figura 2.4: Rastro deixado pela bola se não limparmos a tela a cada ciclo da animação

> Temos uma implicação importante: todo objeto que quiser participar do loop de animação (ou seja, que quiser *ser um sprite*), terá que implementar os métodos `atualizar` e `desenhar`. Este é o conceito de *interface* da Orientação a objetos: um conjunto de métodos que uma classe deve implementar para poder interagir com outra.

Para chamar o próximo ciclo, não podemos simplesmente fazer:

```
requestAnimationFrame(this.proximoFrame);
```

A função de animação é chamada pelo JavaScript como uma função comum, não como um método de objeto — não podemos usar o `this` dentro dela, e fizemos isso várias vezes! Juro, quando fui fazer isto, quebrei a cabeça para descobrir por que não funcionava. A solução é referenciar o objeto em uma variável e chamar uma função anônima, que por sua vez chama `proximoFrame` como um verdadeiro *método* do objeto:

```
var animacao = this;
requestAnimationFrame(function() {
   animacao.proximoFrame();
});
```

Temos agora que implementar o método `limparTela`:

```
limparTela: function() {
   var ctx = this.context;  // Só para facilitar a escrita ;)
   ctx.clearRect(0, 0, ctx.canvas.width, ctx.canvas.height);
}
```

Este método requer um objeto `context` do HTML5 Canvas, portanto vamos recebê-lo no construtor:

```
function Animacao(context) {
   this.context = context;
   this.sprites = [];
   this.ligado = false;
}
```

Temos então que passá-lo no código do aplicativo:

```
// Criando o loop de animação
var animacao = new Animacao(context);
```

> Procure fazer com que uma classe receba de fora e guarde em atributos tudo que ela precisa para executar seu trabalho, de forma que ela fique completamente desacoplada do meio externo. Depender diretamente da variável `context` do aplicativo é mau negócio! Este princípio chama-se *injeção de dependências*.

2.4 Implemente a classe Bola

Esta classe implementará a interface de sprites, portanto deverá ter os métodos `atualizar` e `desenhar`:

```
// Arquivo: bola.js
function Bola(context) {
   this.context = context;
   this.x = 0;
   this.y = 0;
   this.velocidadeX = 0;
   this.velocidadeY = 0;

   // Atributos de desenho padrão
   this.cor = 'black';
   this.raio = 10;
}
Bola.prototype = {
   atualizar: function() {

   },
   desenhar: function() {

   }
}
```

Em `atualizar`, colocamos o algoritmo que quica a bola nas bordas do Canvas. Repare que sempre temos que descontar o raio da bola, pois as coordenadas *x* e *y* se referirão ao seu centro (figura 2.5). Se o centro estiver fora do limite estabelecido, invertemos o sinal da velocidade (multiplicamos por -1) para que a bola passe a ir em sentido contrário, tanto na vertical quanto na horizontal:

```
atualizar: function() {
   var ctx = this.context;

   if (this.x < this.raio || this.x >
         ctx.canvas.width - this.raio)
      this.velocidadeX *= -1;

   if (this.y < this.raio || this.y >
         ctx.canvas.height - this.raio)
      this.velocidadeY *= -1;
```

```
    this.x += this.velocidadeX;
    this.y += this.velocidadeY;
},
```

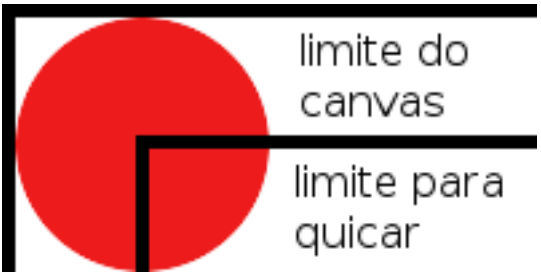


Figura 2.5: A posição da bola é calculada pelo seu centro, portanto os limites para quicá-la têm que descontar o raio.

Em `desenhar`, usamos o método `arc` do `context`. Também usamos os métodos `save` e `restore` para poder configurá-lo com a cor da bola e depois retorná-lo ao estado original. Caso não se lembre destes métodos, reveja a seção 1.2.

```
desenhar: function() {
    var ctx = this.context;

    // Guardar configurações atuais do contexto
    ctx.save();

    // Configurar o contexto de acordo com a bola
    ctx.fillStyle = this.cor;

    // Desenhar
    ctx.beginPath();
    ctx.arc(this.x, this.y, this.raio, 0, 2 * Math.PI, false);
    ctx.fill();

    // Voltar às configurações anteriores
    ctx.restore();
}
```

O aplicativo está pronto! Abra-o no navegador. Se estiver usando o Chrome, pressione `Ctrl+Shift+J` e digite no Console (para fazer isso no Firefox, você deve instalar um plugin como o Firebug):

```
animacao.desligar();
```

A animação para! Agora faça:

```
animacao.ligar();
```

Não é fantástico? Como exercício, crie mais objetos `Bola` com tamanhos, cores e velocidades diferentes, e acrescente-os à animação. Você pode fazer isso no arquivo HTML ou diretamente no Console!

Aqui nós criamos uma primeira base para nosso jogo, que é o loop de animação. No próximo capítulo, vamos aprender como é possível criar a interação do usuário com o jogo.

Capítulo 3

A interação com o jogador — leitura apurada do teclado

Neste tópico, vamos tratar da interação do usuário com o jogo. Tratar eventos de teclado em JavaScript não é uma tarefa difícil, no entanto, há aspectos que precisam ser considerados, em se tratando de jogos.

Faremos uma revisão sobre os eventos de teclado e, em seguida, trataremos dois tipos especiais de interações:

- *o jogador mantém uma tecla pressionada:* muito comum com as setas de movimentação, mas não se limita a estas. O jogo deve responder de determinada maneira (por exemplo, movimentando o herói) enquanto a tecla está pressionada e parar quando o jogador solta a tecla;
- *o jogador pressiona e solta rapidamente a tecla:* são as *teclas de disparo*,

com as quais a ação deve ocorrer uma única vez, no momento exato do pressionamento. Se a tecla continuar pressionada, o jogo deve ignorar seu estado.

Isto tudo parece muito óbvio mas, como veremos, o modelo de eventos do teclado em JavaScript é voltado para a *digitação de caracteres* (lembre-se, a linguagem foi criada para páginas da web) e não nos permite tratar interações comuns em jogos com apenas um comando. Porém, a coisa também não é tão complicada, requerendo apenas alguma codificação manual.

3.1 EventListeners e os eventos keydown e keyup

Qualquer elemento em uma página da web pode ter "ouvintes" (*listeners*) de eventos. No caso dos eventos do teclado, o elemento deve ter o *foco*:

Figura 3.1: Cursor do teclado piscando no campo de busca. O foco do teclado pertence a ele neste momento.

Figura 3.2: Após alguns toques na tecla Tab, o foco vai para o botão *Pesquisa Google* e este fica destacado. O browser aguarda o usuário teclar Enter ou espaço para acionar o botão.

Um *listener* é uma função JavaScript que executa uma ação em resposta a um evento. Essa função pode receber um parâmetro que é preenchido automaticamente com dados sobre o evento ocorrido.

Para associar um listener a um elemento da página, usamos o método *addEventListener*:

```
// Exemplo teórico
var campo = document.getElementById('campo');
campo.addEventListener('nomedoevento',
   function(parametroEvento) {
      // Ações de resposta ao evento vão aqui
   });
```

Em JavaScript, são três os eventos associados ao teclado:

- **keydown**: ocorre quando uma tecla é pressionada (abaixada). Se o usuário a mantiver segurando, o evento é disparado repetidamente.
- **keypress**: ocorre logo após *keydown*, no caso de a tecla ser de caractere. Se a tecla continuar pressionada, o browser dispara os eventos alternadamente: keydown-keypress-keydown-keypress-etc.
- **keyup**: ocorre uma única vez, quando uma tecla é solta.

Figura 3.3: Tecla mantida pressionada. O computador produz um caractere, faz uma pausa, e dispara outros em sequência.

> Para o desenvolvimento de jogos, somente os eventos *keydown* e *keyup* serão necessários. Tudo o que nós precisaremos saber serão os momentos quando uma tecla é pressionada e quando é solta, para poder disparar ações ao pressionar ou detectar se uma tecla está ou não pressionada. O evento *keypress* é específico para tratar digitação de caracteres.

Seria tudo muito maravilhoso se pudéssemos codificar nossas ações diretamente no evento *keydown*! No entanto, o resultado ficaria horrível. O modelo de eventos para a digitação de caracteres (figura 3.3) não serve para movimentar personagens. Já vi alguns joguinhos de tutoriais na internet serem feitos dessa maneira. Faça um teste e comprove por si mesmo:

```
<!-- arquivo teclado-teste-1.html -->
<!DOCTYPE html>
<html>

<head>
   <title>Movimentando personagem com keydown</title>
</head>

<body>
   <canvas id="canvas_teclado_1" width="500" height="500">
   </canvas>
   <script>
      // Interação com o teclado vai aqui
   </script>
</body>
```

```
</html>
```

Agora iremos criar um *listener* para o evento `keydown`, que desloca um "personagem" para a direita ou para a esquerda. As teclas de seta são reconhecidas através da propriedade `keyCode` do objeto `evento`, recebido como parâmetro.

Na tag `<script>`, que vem logo após o Canvas, digite:

```
// Referências do Canvas
var canvas = document.getElementById('canvas_teclado_1');
var context = canvas.getContext('2d');

// Posição inicial do personagem
var posicao = 0;
desenharPersonagem();

document.addEventListener('keydown', function(evento) {
   if (evento.keyCode == 37) {  // Seta para esquerda
      posicao -= 10;
      desenharPersonagem();
   }
   else if (evento.keyCode == 39) {  // Seta para direita
      posicao += 10;
      desenharPersonagem();
   }
});

// Um personagem não muito simpático, mas...
function desenharPersonagem() {
   context.clearRect(0, 0, canvas.width, canvas.height);
   context.fillRect(posicao, 100, 20, 50);
}
```

Abra a página no browser e mova o retângulo com as setas direita e esquerda. Perceba que, ao iniciar o movimento para algum lado, ocorre uma pausa após o primeiro deslocamento, para depois ele ocorrer normalmente.

Figura 3.4: Nosso herói parece que está manco...

Não ficou bom, não é verdade? Mas podemos aprender muitas coisas debruçando-se sobre este trecho:

```
document.addEventListener('keydown', function(evento) {
   if (evento.keyCode == 37) {  // Seta para esquerda
      posicao -= 10;
      desenharPersonagem();
   }
   else if (evento.keyCode == 39) {  // Seta para direita
      posicao += 10;
      desenharPersonagem();
   }
});
```

- Primeiro, quero ouvir eventos do teclado ocorridos em todo o documento. Portanto, adiciono o listener ao elemento `document`.
- A `function` recebeu um parâmetro de nome `evento` (posso dar qualquer nome para ele). Esse parâmetro recebeu do JavaScript automaticamente um objeto com propriedades relacionadas ao evento.
- A propriedade `keyCode` desse objeto contém um valor único para cada tecla. Não é difícil descobrir os códigos fazendo uma rápida pesquisa na internet, ou criando um pequeno programa para esse fim.

3.2 Detectando se uma tecla está ou não pressionada

Como aprendemos (e comprovamos, caso você tenha digitado o código anterior), o modelo de eventos de teclado do JavaScript é eficaz para tratar digitação, mas completamente bizarro para jogos. Como não há nenhum comando para eu perguntar: "a tecla *tal* está pressionada?", tenho que manter um registro de teclas pressionadas manualmente. Para facilitar, seguiremos o modelo de criar uma classe para tratar deste problema.

Teste da classe `Teclado`

Criemos uma página semelhante à anterior, mas onde nosso corajoso herói será controlado de maneira mais eficiente. Imagine podermos fazer:

```
// Exemplo teórico
var teclado = new Teclado(document);

if (teclado.pressionada(SETA_DIREITA))
   posicao += 10;
```

Bem, então... façamos! Repare na simplicidade com que queremos controlar o teclado. Delegamos à classe `Teclado` a responsabilidade de saber se determinada tecla está ou não pressionada.

```
<!-- arquivo teste-teclado-2.html -->
<!DOCTYPE html>
<html>

<head>
   <title>Movimentando personagem com classe JavaScript</title>
   <script src="teclado.js"></script>
</head>

<body>
   <canvas id="canvas_teclado_2" width="500" height="500">
   </canvas>
   <script>
```

```
      // Aqui virá um controle de teclado mais apurado
   </script>
</body>

</html>
```

O JavaScript conterá partes bem parecidas com o do exemplo anterior, mas introduzimos aqui um loop de animação. É ele que vai avaliar constantemente o estado de determinadas teclas.

Primeiro referenciamos o Canvas normalmente:

```
var canvas = document.getElementById('canvas_teclado_2');
var context = canvas.getContext('2d');
```

Em seguida, definimos a posição inicial do personagem e mandamos desenhar:

```
var posicao = 0;
desenharPersonagem();
```

Agora vamos criar o controlador do teclado. Quero passar para ele o elemento da página onde os eventos serão ouvidos (no caso, `document`):

```
var teclado = new Teclado(document);
```

O loop de animação será bem básico, por enquanto através de uma simples função. Nesta função, primeiro lemos o estado das teclas que queremos e mudamos a posição do personagem de acordo. Em seguida, nós o desenhamos e chamamos o próximo ciclo:

```
requestAnimationFrame(animar);

function animar() {
   if (teclado.pressionada(SETA_ESQUERDA))
      posicao -= 10;
   else if (teclado.pressionada(SETA_DIREITA))
      posicao += 10;

   desenharPersonagem();
   requestAnimationFrame(animar);
}
```

A função `desenharPersonagem` é a mesma criada anteriormente:

```
// Nosso herói continua esbanjando charme! :D
function desenharPersonagem() {
    context.clearRect(0, 0, canvas.width, canvas.height);
    context.fillRect(posicao, 100, 20, 50);
}
```

A classe `Teclado`

Esta classe recebe um elemento qualquer da página no construtor e atribui a ele os eventos que vão registrar o estado das teclas em um array. O método `pressionada` devolve o valor `true` caso a tecla conste como pressionada no momento:

```
// arquivo: teclado.js
// Códigos de teclas - aqui vão todos os que forem necessários
var SETA_ESQUERDA = 37;
var SETA_DIREITA = 39;

function Teclado(elemento) {
    this.elemento = elemento;

    // Array de teclas pressionadas
    this.pressionadas = [];

    // Registrando o estado das teclas no array
    var teclado = this;
    elemento.addEventListener('keydown', function(evento) {
        teclado.pressionadas[evento.keyCode] = true;
    });
    elemento.addEventListener('keyup', function(evento) {
        teclado.pressionadas[evento.keyCode] = false;
    });
}
Teclado.prototype = {
    pressionada: function(tecla) {
        return this.pressionadas[tecla];
    }
}
```

Abra a página e veja que o "personagem" move-se com muito mais suavidade e fluência!

3.3 Efetuando disparos — detectando somente o primeiro keydown

Vamos dar um superpoder ao herói: atirar. Neste ponto, para facilitar a nossa vida, vamos usar a classe `Animacao` e o conceito de sprites aprendidos anteriormente.

Que tal poder mandar o herói atirar da forma mostrada a seguir? Se a classe `Teclado` sabe se uma tecla está pressionada, ela também pode saber disparar uma ação única no momento do pressionamento!

```
// Exemplo teórico
teclado.disparou(ESPACO, function() {
   heroi.atirar();
});
```

Figura 3.5: Herói atirando

Vamos criar uma nova página. Os scripts `bola.js` e `animacao.js` são os mesmos já criados. Certifique-se de ter uma cópia deles na pasta deste teste (ou linká-los corretamente em seu computador).

```
<!-- arquivo: teste-teclado-3.html -->
<!DOCTYPE html>
<html>

<head>
   <title>Um herói que atira</title>
   <script src="teclado.js"></script>
   <script src="heroi.js"></script>
   <script src="bola.js"></script>
   <script src="animacao.js"></script>
</head>

<body>
   <canvas id="canvas_teclado_3" width="500" height="500">
   </canvas>
   <script>
      // Isto vai começar a parecer um jogo!
   </script>
</body>

</html>
```

Veja como o script do teste é simples e conciso. Nesta altura, eu espero sinceramente que você esteja entendendo o porquê de estarmos fazendo dessa forma. Criar uma nova ação, um novo personagem, um novo inimigo deverá ser cada vez mais simples a partir de agora:

```
var canvas = document.getElementById('canvas_teclado_3');
var context = canvas.getContext('2d');

var teclado = new Teclado(document);
var animacao = new Animacao(context);

// Um sprite pode ler o teclado para saber
// como se comportar
var heroi = new Heroi(context, teclado);
heroi.x = 0;
heroi.y = 100;
animacao.novoSprite(heroi);
```

```
teclado.disparou(ESPACO, function() {
   heroi.atirar();
});

animacao.ligar();
```

Precisamos criar o método `disparou` na classe `Teclado`. Para que ele funcione corretamente, é preciso registrar que determinada tecla foi disparada (para que os eventos *keydown* subsequentes não provoquem o disparo novamente). Criamos mais dois arrays, um para registrar os disparos e outro para guardar as funções a serem executadas:

```
// Códigos de teclas - aqui vão todos os que forem necessários
var SETA_ESQUERDA = 37;
var SETA_DIREITA = 39;
var ESPACO = 32;

function Teclado(elemento) {
   this.elemento = elemento;

   // Array de teclas pressionadas
   this.pressionadas = [];

   // Array de teclas disparadas
   this.disparadas = [];

   // Funções de disparo
   this.funcoesDisparo = [];

   // continua...
}
```

No evento *keydown*, verificamos se a tecla pressionada possui uma função de disparo associada e se seu disparo já foi processado. A instrução `this.funcoesDisparo[tecla] () ;` executa a função guardada no array devido aos parênteses no final:

```
var teclado = this;
```

```
elemento.addEventListener('keydown', function(evento) {
    var tecla = evento.keyCode;  // Tornando mais legível ;)
    teclado.pressionadas[tecla] = true;

    // Disparar somente se for o primeiro keydown da tecla
    if (teclado.funcoesDisparo[tecla] &&
        ! teclado.disparadas[tecla]) {

        teclado.disparadas[tecla] = true;
        teclado.funcoesDisparo[tecla] () ;
    }
});
```

No evento *keyup* setamos o estado "disparada" para `false`, de modo a tornar possíveis novos disparos:

```
elemento.addEventListener('keyup', function(evento) {
    teclado.pressionadas[evento.keyCode] = false;
    teclado.disparadas[evento.keyCode] = false;
});
```

Enfim, o método `disparou` necessita apenas guardar uma função de disparo para uma tecla. O evento *keydown* está cuidando de tudo!

```
disparou: function(tecla, callback) {
    this.funcoesDisparo[tecla] = callback;
}
```

Em caso de dúvidas, veja o novo código completo da classe `Teclado`:

```
// arquivo teclado.js - versão final
// Códigos de teclas - aqui vão todos os que forem necessários
var SETA_ESQUERDA = 37;
var SETA_DIREITA = 39;
var ESPACO = 32;

function Teclado(elemento) {
    this.elemento = elemento;

    // Array de teclas pressionadas
```

```
    this.pressionadas = [];

    // Array de teclas disparadas
    this.disparadas = [];

    // Funções de disparo
    this.funcoesDisparo = [];

    var teclado = this;

    elemento.addEventListener('keydown', function(evento) {
        var tecla = evento.keyCode;  // Tornando mais legível ;)
        teclado.pressionadas[tecla] = true;

        // Disparar somente se for o primeiro keydown da tecla
        if (teclado.funcoesDisparo[tecla] &&
            ! teclado.disparadas[tecla]) {

            teclado.disparadas[tecla] = true;
            teclado.funcoesDisparo[tecla] () ;
        }
    });

    elemento.addEventListener('keyup', function(evento) {
        teclado.pressionadas[evento.keyCode] = false;
        teclado.disparadas[evento.keyCode] = false;
    });
}
Teclado.prototype = {
    pressionada: function(tecla) {
        return this.pressionadas[tecla];
    },
    disparou: function(tecla, callback) {
        this.funcoesDisparo[tecla] = callback;
    }
}
```

É bastante abstrato, mas se conseguirmos programar de maneira mais geral agora, teremos um game engine bastante robusto e prático para criar

novos jogos de maneira extremamente veloz!

Crie a classe `Heroi` como um sprite

Para nosso código funcionar, temos que criar o sprite do herói. Como o sprite é responsável pelo seu próprio comportamento, ele deve receber o objeto que controla o teclado para poder decidir o que vai fazer em cada ciclo da animação.

O herói também deve ter o método `atirar`, chamado no teste.

```
// arquivo: heroi.js
function Heroi(context, teclado) {
   this.context = context;
   this.teclado = teclado;
   this.x = 0;
   this.y = 0;
}
Heroi.prototype = {
   atualizar: function() {

   },
   desenhar: function() {

   },
   atirar: function() {

   }
}
```

No método `atualizar`, lemos o estado do teclado e movemos o personagem de acordo. Também não custa nada impedir que ultrapasse as bordas da tela:

```
atualizar: function() {
   if (this.teclado.pressionada(SETA_ESQUERDA) && this.x > 0)
      this.x -= 10;
   else if (this.teclado.pressionada(SETA_DIREITA) &&
            this.x < this.context.canvas.width - 20)
      this.x += 10;
},
```

No método desenhar... bem, eu vou continuar mantendo as coisas simples e fazer um simples retângulo. No capítulo sobre **spritesheets** (4) a coisa vai ficar mais interessante, prometo!

```
desenhar: function() {
   this.context.fillRect(this.x, this.y, 20, 50);
},
```

Agora, a parte mais interessante: fazê-lo atirar. Neste ponto, devemos ler novamente o estado do teclado para saber para que lado o tiro vai. Vamos aproveitar a classe `Bola` que já temos:

```
atirar: function() {
   var tiro = new Bola(this.context);
   tiro.x = this.x + 10;
   tiro.y = this.y + 10;
   tiro.raio = 2;
   tiro.cor = 'red';

   if (this.teclado.pressionada(SETA_ESQUERDA))
      tiro.velocidadeX = -20;
   else
      tiro.velocidadeX = 20;

   // Não tenho como incluir nada na animação!
   this.animacao.novoSprite(tiro);
}
```

Como o herói vai acrescentar um tiro à animação sem uma referência a ela? Vamos recebê-la também no construtor:

```
function Heroi(context, teclado, animacao) {
   this.context = context;
   this.teclado = teclado;
   this.animacao = animacao;
   this.x = 0;
   this.y = 0;
}
```

E passá-la no aplicativo de teste:

```
var heroi = new Heroi(context, teclado, animacao);
```

Faça o teste. Se você fez tudo correto, o herói deve atirar ao teclarmos `espaço`. Mas ainda temos dois pequenos problemas:

1) a classe Bola está programada para quicar, portanto não vemos os tiros sumirem da tela (figura 3.6);

2) quando parado, o herói atira sempre para a direita.

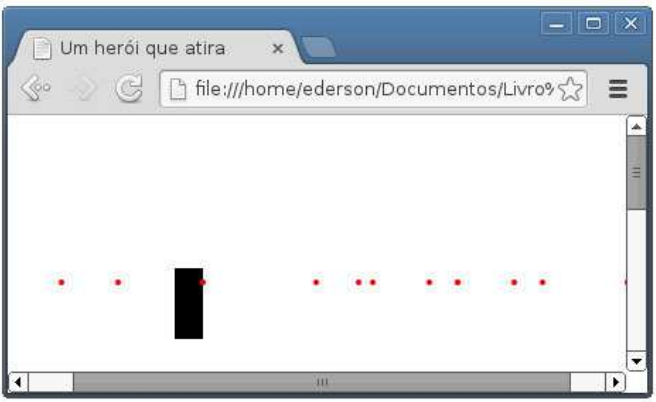


Figura 3.6: Os tiros ficam quicando na tela, ao invés de sumirem

Para resolver o primeiro problema, vamos mudar o algoritmo do método `atualizar` da classe `Bola`. Recomendo que este teste referencie sua própria cópia do arquivo `bola.js`, para não alterar o comportamento dos testes anteriores.

```
atualizar: function() {

   // Tiramos os testes que quicam a bola na borda do Canvas

   this.x += this.velocidadeX;
   this.y += this.velocidadeY;
},
```

Quanto à direção do tiro, criamos no `Heroi` um atributo que guarda a sua direção:

```
// Códigos únicos para as direções
var DIRECAO_ESQUERDA = 1;
var DIRECAO_DIREITA = 2;

function Heroi(context, teclado, animacao) {
   this.context = context;
   this.teclado = teclado;
   this.animacao = animacao;
   this.x = 0;
   this.y = 0;

   // Direção padrão
   this.direcao = DIRECAO_DIREITA;
}
```

Desta forma, ao movimentar o herói, guardamos a direção em que ele se encontra nesse atributo:

```
atualizar: function() {
   if (this.teclado.pressionada(SETA_ESQUERDA)
         && this.x > 0) {
      this.direcao = DIRECAO_ESQUERDA;
      this.x -= 10;
   }
   else if (this.teclado.pressionada(SETA_DIREITA) &&
            this.x < this.context.canvas.width - 20) {
      this.direcao = DIRECAO_DIREITA;
      this.x += 10;
   }
},
```

Por fim, basta ler essa direção ao soltar o tiro, em vez do estado de determinada tecla:

```
atirar: function() {
   // ...
```

```
    // Lendo a direção atual
    if (this.direcao == DIRECAO_ESQUERDA)
        tiro.velocidadeX = -20;
    else
        tiro.velocidadeX = 20;

    // ...
}
```

Pronto! Temos um herói que se movimenta e atira dentro de um loop de animação. Embora ainda esteja muito básico, é preciso que os conceitos aprendidos fiquem bem fixados para seguirmos para as próximas etapas. Procure reler e refazer os códigos de cada capítulo diversas vezes. **Tente também fazer algumas coisas diferentes, como mover ou atirar na vertical!** Para isso, os códigos de teclas são 38 (acima) e 40 (abaixo), e as alterações na posição *y* dos sprites serão as mesmas já feitas para *x*, somente incrementando a partir da `velocidadeY`. Experimente!

No próximo capítulo, introduzirei as folhas de sprites (*spritesheets*), que nos permitirão animar nosso herói a partir de imagens.

Capítulo 4

Folhas de sprites — spritesheets

4.1 Conceito e abordagem utilizada

Um pequeno *game engine* está começando a tomar forma. Já temos um modo robusto de controlar o loop de animação e capturar a entrada do usuário no teclado de uma maneira eficiente para jogos.

Neste capítulo, avançaremos um pouco além e faremos algo muito interessante e fundamental: as *spritesheets* (folhas de sprites). Um sprite, como você aprendeu no capítulo 2.1, é cada elemento controlado pelo loop de animação (o herói, um inimigo, um bloco ou plataforma etc.). Uma folha de sprites é uma imagem contendo várias partes de uma animação. Essas partes são alternadas constantemente para produzir o movimento de um ou mais sprites.

Figura 4.1: Folha de sprites (spritesheet)

É uma boa prática carregar imagens maiores contendo uma porção de outras menores, em vez de ter cada pequena imagem em seu próprio arquivo. Lembre-se de que, em ambiente web, cada imagem requer nova conexão do browser com o servidor.

Programar animações com spritesheets pode ser tão simples ou tão trabalhoso quanto você desejar ou precisar, dependendo da abordagem utilizada. No livro Core HTML5 Canvas[1], de David Geary, as posições de cada figurinha (x, y, largura e altura) dentro da spritesheet são chumbadas em grandes arrays:

```
var sprite = [ [0, 71, 90, 80],
               [40, 71, 80, 85], ... ]; // Continua...
```

Essa abordagem é bastante econômica em termos do espaço ocupado pelas imagens, mas gera muito trabalho extra para coletar a posição de cada imagem e, principalmente, para enquadrá-las adequadamente. Aqui, adotaremos uma abordagem bastante sistematizada:

- cada spritesheet será dividida em espaços iguais, por isso as imagens devem estar corretamente distribuídas;
- cada linha da spritesheet corresponderá a um *estado* do sprite (ex.: parado, correndo para a direita, correndo para a esquerda);
- a animação de um estado usará somente as figuras na sua linha própria.

Isso gera grande quantidade de espaço vazio na imagem, aumentando o tamanho do arquivo, mas certamente facilitará muito a programação. Tenha em mente que esta não é a única abordagem válida, mas para fins didáticos é a que iremos utilizar V convenhamos, o modo mais econômico descrito no livro Core HTML5 Canvas é totalmente improdutivo, em especial quando o foco é o aprendizado das técnicas de programação, não de enquadramento de imagens.

Figura 4.2: Spritesheet dividida em partes iguais e organizando os estados por linhas

O CorelDraw possui a ferramenta *Papel Gráfico*, que cria um quadriculado dividindo uma área em partes iguais. Neste programa e também no Photoshop, Illustrator e em outros, podemos usar as réguas e linhas-guia. Se você não conhece estes recursos, sugiro que pesquise a respeito. Sugestões:

http://coreldrawtips.com/site/using-the-graph-paper-tool-in-coreldraw
http://helpx.adobe.com/br/photoshop/using/grid-guides.html

4.2 Carregando imagens e fazendo recortes (clipping)

Carregar uma imagem do servidor pelo JavaScript é bastante simples. Primeiro instanciamos um objeto `Image`, em seguida, setamos seu atributo `src` para o caminho da imagem:

```
var imgSonic = new Image();
imgSonic.src = 'spritesheet.png';
```

A imagem possui o evento `onload`, que é disparado quando o carregamento da imagem está completo:

```
imgSonic.onload = function() { ... }
```

Quando tivermos muitas imagens, o evento onload de cada uma pode incrementar uma porcentagem ou barrinha de "carregando". Isso ficará para outra hora!

Vamos agora fazer o *clipping* (enquadramento), que é o recorte da parte exata da spritesheet que queremos. Por exemplo, suponha que queremos a figurinha na linha 2, coluna 7 (contados a partir de zero). Na hora de programar as animações, ficará muito mais fácil expressar dessa maneira. No entanto, precisaremos calcular a posição (x, y) do recorte da imagem.

```
<!-- arquivo: clipping.html -->
<!DOCTYPE html>
<html>

<head>
   <title>
      Spritesheet - recorte e enquadramento (clipping)
   </title>
</head>

<body>
   <canvas id="canvas_clipping" width="500" height="500">
   </canvas>
   <script>
```

```
        var imgSonic = new Image();
        imgSonic.src = 'spritesheet.png';
        imgSonic.onload = function() {
            // Aqui faremos o clipping!
        }
    </script>
</body>

</html>
```

No evento `onload` da imagem, calcularemos as posições *x* e *y* do recorte. Como estamos considerando quadros de tamanhos iguais, basta dividir a largura total pelo número de colunas para obter a largura de um quadro. Depois basta multiplicar este valor pela coluna desejada, e temos a posição x onde o recorte se inicia. Para a posição y, o processo é análogo a partir da altura total e número de linhas:

```
imgSonic.onload = function() {
    // Passo estes valores conforme a minha spritesheet
    var linhas = 3;
    var colunas = 8;

    // Dimensão de cada quadro
    var largura = imgSonic.width / colunas;
    var altura = imgSonic.height / linhas;

    // Quadro que eu quero (expresso em linha e coluna)
    var queroLinha = 2;
    var queroColuna = 7;

    // Posição de recorte
    var x = largura * queroColuna;
    var y = altura * queroLinha;

    // Continua...
}
```

Tendo calculado x, y, largura e altura, basta lembrarmos do método `drawImage` do *context* (veja o capítulo 1.2). Este método pode ser chamado

de duas maneiras:

- Desenhando uma imagem inteira:

```
context.drawImage(imagem, x, y, largura, altura);
```

- Fazendo clipping:

```
context.drawImage(
   imagem,
   xOrigem,
   yOrigem,
   larguraOrigem,
   alturaOrigem,
   xDestino,
   yDestino,
   larguraDestino,
   alturaDestino
);
```

Finalizemos, então, o nosso código:

```
var context =
    document.getElementById('canvas_clipping').getContext('2d');
context.drawImage(
   imgSonic,
   x,
   y,
   largura,
   altura,
   100, // Posição no canvas onde quero desenhar
   100,
   largura,
   altura
);
```

Veja o resultado no browser:

Figura 4.3: Fazendo clipping na spritesheet (quadro na linha 2, coluna 7)

4.3 Animações de sprite — a classe Spritesheet

Daremos prosseguimento à confecção do nosso game engine. A classe `Spritesheet` será responsável por:

- Avançar a animação, escolhendo qual o quadro a ser desenhado no momento;
- Calcular as posições de recorte e realizar o clipping, dados os números de linhas e colunas;
- Gerenciar o tempo entre um quadro e outro.

Como de costume, escreveremos primeiro um código de teste, com o qual tentamos manter a interface da classe o mais enxuta possível. Inicialmente, faremos o Sonic correr para a direita. Cada quadro terá a duração de 60 milissegundos, podendo ser ajustado posteriormente.

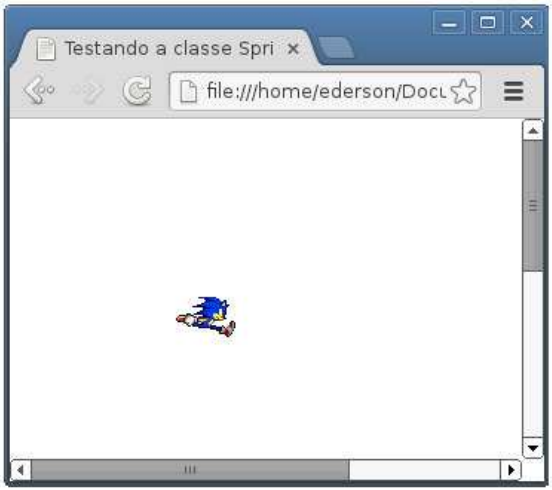


Figura 4.4: Sonic correndo na tela

Dessa forma, a classe `Spritesheet` deverá alternar para nós os quadros da segunda linha (índice 1, contado a partir de zero — veja figura 4.2). Ainda não usaremos a classe `Animacao`, apenas uma função simples que servirá como game loop.

```
<!-- arquivo: teste-spritesheet-1.html -->
<!DOCTYPE html>
<html>

<head>
   <title>Testando a classe Spritesheet</title>
   <script src="spritesheet.js"></script>
</head>

<body>
   <canvas id="canvas_spritesheet_1" width="500" height="500">
   </canvas>
   <script>
      // Vamos tentar mandar na Spritesheet
      // da maneira mais simples!
```

```
    </script>
</body>

</html>
```

No JavaScript, queremos dizer à classe `Spritesheet` os números de linhas e colunas, e escolhemos qual linha queremos animar. As colunas deverão ser alternadas automaticamente.

Também passamos no construtor o *context* do Canvas e a imagem a ser desenhada:

```
var context = document.getElementById('canvas_spritesheet_1')
              .getContext('2d');

var imgSonic = new Image();
imgSonic.src = 'spritesheet.png';

// Quero passar: context, imagem, linhas, colunas
var sheet = new Spritesheet(context, imgSonic, 3, 8);

// Duração de cada quadro
sheet.intervalo = 60;

// "correndo para direita"
sheet.linha = 1;

// Animação
imgSonic.onload = gameLoop;
```

Na função `gameLoop`, ordeno à `Spritesheet` que avance um quadro e que desenhe o quadro atual na posição que eu mandar:

```
function gameLoop() {
    context.clearRect(0, 0, context.canvas.width,
          context.canvas.height);

    // Avançar na animação
    sheet.proximoQuadro();
```

```
    // Onde desenhar o quadro atual
    sheet.desenhar(100, 100);

    requestAnimationFrame(gameLoop);
}
```

Teste pronto, passemos ao esboço da classe `Spritesheet`:

```
function Spritesheet(context, imagem, linhas, colunas) {
    this.context = context;
    this.imagem = imagem;
    this.numLinhas = linhas;
    this.numColunas = colunas;
    this.intervalo = 0;
    this.linha = 0;
    this.coluna = 0;
}
Spritesheet.prototype = {
    proximoQuadro: function() {

    },
    desenhar: function(x, y) {

    }
}
```

O cálculo do quadro atual é bem simples, bastando incrementar a coluna atual e voltar para zero quando exceder o número de colunas:

```
proximoQuadro: function() {
    if (this.coluna < this.numColunas - 1)
        this.coluna++;
    else
        this.coluna = 0;
},
```

Mas há um porém: como temos um **intervalo** de tempo definido para a mudança de quadro, temos que levá-lo em conta! Isso não é difícil: vamos manter um registro da última mudança e, a cada ciclo, verificar se já passou o tempo especificado:

```
proximoQuadro: function() {
   // Momento atual
   var agora = new Date().getTime();

   // Se ainda não tem último tempo medido
   if (! this.ultimoTempo) this.ultimoTempo = agora;

   // Já é hora de mudar de coluna?
   if (agora - this.ultimoTempo < this.intervalo) return;

   if (this.coluna < this.numColunas - 1)
      this.coluna++;
   else
      this.coluna = 0;

   // Guardar hora da última mudança
   this.ultimoTempo = agora;
},
```

No método `desenhar`, basta fazer o clipping como aprendemos:

```
desenhar: function(x, y) {
   var larguraQuadro = this.imagem.width / this.numColunas;
   var alturaQuadro = this.imagem.height / this.numLinhas;

   this.context.drawImage(
      this.imagem,
      larguraQuadro * this.coluna,
      alturaQuadro * this.linha,
      largura,
      altura,
      x,
      y,
      largura,
      altura
   );
}
```

Neste ponto, já temos o Sonic correndo na tela, mas ainda sem alterar sua posição.

4.4 Controle o herói pelo teclado e veja sua animação

Chegou a hora de juntar a `Spritesheet` com as outras classes criadas anteriormente (`Animacao` e `Teclado`). Vamos controlar o Sonic correndo!

O sprite do Sonic terá sa funções de:

- Configurar sua respectiva spritesheet;
- Ler o estado das teclas de seta e mudar o estado do sprite;
- Selecionar a linha na spritesheet correspondente ao estado desejado;
- Responder ao loop de animação, implementando os velhos métodos `atualizar` e `desenhar`.

```
<!-- arquivo: teste-spritesheet-2.html -->
<!DOCTYPE html>
<html>

<head>
   <title>Personagem controlável e animado</title>
   <script src="spritesheet.js"></script>
   <script src="animacao.js"></script>
   <script src="teclado.js"></script>
   <script src="sonic.js"></script>
</head>

<body>
   <canvas id="canvas_sonic" width="500" height="500"></canvas>
   <script>
      // Mover um personagem nunca foi tão fácil!
   </script>
</body>

</html>
```

Primeiro, façamos as referências do Canvas, como de costume. Depois, criamos os controles de teclado, de animação e o Sonic. Este precisará ler o

teclado para movimentar-se. Espero que você esteja percebendo que praticamente tudo que criaremos seguirá este padrão:

```
var canvas = document.getElementById('canvas_sonic');
var context = canvas.getContext('2d');

var teclado = new Teclado(document);
var animacao = new Animacao(context);

var sonic = new Sonic(context, teclado);
sonic.x = 0;
sonic.y = 200;
animacao.novoSprite(sonic);

animacao.ligar();
```

Crie a classe `Sonic`

Pelos comandos dados ao Sonic anteriormente, seu esqueleto conterá somente o `context`, o teclado, a posição (x, y) e os métodos de sprite:

```
// arquivo: sonic.js
function Sonic(context, teclado) {
   this.context = context;
   this.teclado = teclado;
   this.x = 0;
   this.y = 0;
}
Sonic.prototype = {
   atualizar: function() {

   },
   desenhar: function() {

   }
}
```

Uma das funções da classe Sonic é controlar sua spritesheet, portanto vamos criá-la no construtor. Optei por sempre receber as imagens pelo construtor, pois assim o aplicativo principal terá controle sobre seu evento `onload`

(por exemplo, para animar uma tela de loading que será criada posteriormente).

```
function Sonic(context, teclado, imagem) {
   this.context = context;
   this.teclado = teclado;
   this.x = 0;
   this.y = 0;

   // Criando a spritesheet a partir da imagem recebida
   this.sheet = new Spritesheet(context, imagem, 3, 8);
   this.sheet.intervalo = 60;
}
```

Arrumemos o teste para fornecer a imagem:

```
var imgSonic = new Image();
imgSonic.src = 'spritesheet.png';

var sonic = new Sonic(context, teclado, imgSonic);
```

Vamos também ligar a animação no `onload`:

```
// No final do script
imgSonic.onload = function() {
   animacao.ligar();
}
```

Estados do sprite

Já sabemos movimentar um sprite na tela a partir do teclado. No entanto, o sprite do Sonic só deverá ser animado se alguma seta estiver pressionada; do contrário, a animação de sprites não deve ocorrer. E ele deve ser animado na direção correta. Aí entra a necessidade de termos bem definidos quais os *estados* possíveis para o sprite `Sonic`.

> Na teoria da Orientação a objetos, um *estado* é uma forma em que um objeto se encontra e que determina seu comportamento. Em estados diferentes, a mesma mensagem (chamada de método) enviada a um objeto pode produzir comportamentos diferentes.
>
> Os estados são definidos a partir de um ou mais atributos de um objeto. Por exemplo, uma conta bancária pode estar em um estado `NO_VERMELHO` caso o saldo seja menor que zero.

O estado do Sonic dependerá de dois atributos: `direcao` e `andando`. Estando parado ou andando, ele pode estar virado para a direita ou para a esquerda. Dessa forma, temos quatro estados possíveis para selecionar as imagens e animar (ou não) a partir da spritesheet.

Vamos definir os as direções do Sonic com valores únicos e iniciar o estado atual no construtor:

```
var SONIC_DIREITA = 1;
var SONIC_ESQUERDA = 2;

function Sonic(context, teclado, imagem) {
   this.context = context;
   this.teclado = teclado;
   this.x = 0;
   this.y = 0;

   // Criando a spritesheet a partir da imagem recebida
   this.sheet = new Spritesheet(context, imagem, 3, 8);
   this.sheet.intervalo = 60;

   // Estado inicial
   this.andando = false;
   this.direcao = SONIC_DIREITA;
}
```

Quero agora testar a seta para direita e movimentar o Sonic de acordo. Caso o Sonic já não esteja nesse estado, eu tenho que iniciar a respectiva animação na spritesheet. Ao disparar uma mudança de estado, sempre pode-

mos checar o estado anterior e tomar as medidas necessárias. No método `atualizar`, fazemos:

```
if (this.teclado.pressionada(SETA_DIREITA)) {
   // Se já não estava neste estado...
   if (! this.andando || this.direcao != SONIC_DIREITA) {
      // Seleciono o quadro da spritesheet
      this.sheet.linha = 1;
      this.sheet.coluna = 0;
   }

   // continua ...
}
```

Posicionamos a nossa spritesheet na primeira coluna (zero), da linha relativa ao Sonic indo para a direita. Ainda dentro do primeiro `if`, devemos então definir o estado atual, animar a spritesheet e deslocar o Sonic:

```
if (this.teclado.pressionada(SETA_DIREITA)) {

   // ...

   // Configuro o estado atual
   this.andando = true;
   this.direcao = SONIC_DIREITA;

   // Neste estado, a animação da spritesheet deve rodar
   this.sheet.proximoQuadro();

   // Desloco o Sonic
   this.x += this.velocidade;
}
```

Usamos um novo atributo, `velocidade`. Vamos iniciá-lo com um valor padrão no construtor:

```
// Pode ser ao final do construtor
this.velocidade = 10;
```

O mesmo raciocínio será usado para tratar a movimentação para a esquerda. Continuando o método `atualizar`:

```
else if (this.teclado.pressionada(SETA_ESQUERDA)) {
   if (! this.andando || this.direcao != SONIC_ESQUERDA) {
      this.sheet.linha = 2;  // Atenção, aqui será 2!
      this.sheet.coluna = 0;
   }

   this.andando = true;
   this.direcao = SONIC_ESQUERDA;
   this.sheet.proximoQuadro();
   this.x -= this.velocidade;  // E aqui é sinal de menos!
}
```

Finalmente, se nenhuma tecla de movimentação estiver pressionada, o Sonic está parado. Devemos, no entanto, checar sua direção atual para saber qual imagem do Sonic parado devemos usar, se virado para a esquerda ou para a direita. Também não chamamos o método `proximoQuadro`, pois neste estado não há animação.

```
else {
   if (this.direcao == SONIC_DIREITA)
      this.sheet.coluna = 0;
   else if (this.direcao == SONIC_ESQUERDA)
      this.sheet.coluna = 1;

   this.sheet.linha = 0;
   this.andando = false;
}
```

Todo este processamento está ocorrendo, mas sequer podemos ver o resultado: o método `desenhar` não foi implementado! Aqui, podemos simplesmente chamar o `desenhar` da classe `Spritesheet`:

```
desenhar: function() {
   this.sheet.desenhar(this.x, this.y);
}
```

Isso é tudo! Divirta-se controlando o Sonic para os lados. Ele começa inicialmente parado e virado para a direita, pois assim definimos no construtor, como estado inicial.

No próximo capítulo, falarei sobre detecção de colisões, um aspecto fundamental da grande maioria dos jogos. Seja para golpear um inimigo, ou para coletar argolas, a interação do sprite do personagem com os outros (inimigos, blocos, plataformas) se dará através da colisão entre as áreas que ocupam na tela.

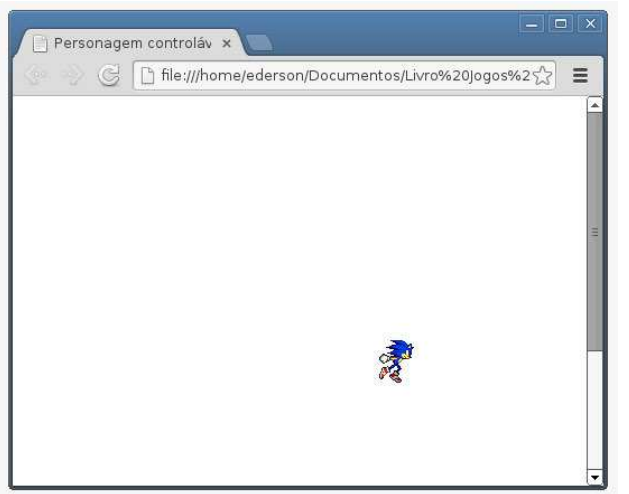


Figura 4.5: Sprite controlado pelo teclado

Capítulo 5

Detecção de colisões

Chegou a hora de detectarmos colisões, ou seja, saber quando um elemento toca outro na tela. Isso tornará possível saber quando um inimigo atacou o herói ou foi atingido, ou quando o herói coletou um item. A partir de colisões, disparamos inúmeros acontecimentos que são a base dos jogos, em especial os de aventura e ação.

5.1 Colisão entre retângulos

Um dos métodos mais simples para detectar colisões é criar uma caixa delimitadora (*bounding box*) ao redor de cada sprite e verificar a intersecção entre os retângulos:

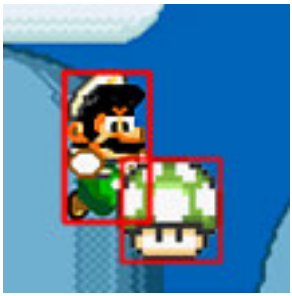


Figura 5.1: Sprites colidindo: retângulos (bounding boxes) apresentam intersecção

Figura 5.2: Sem intersecção, não há colisão

Mas, quem dera que tudo fosse perfeito! Isso pode nos trazer alguns problemas devido às áreas vazias dos sprites:

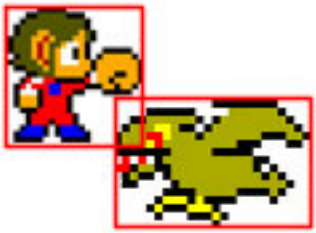


Figura 5.3: Retângulos intersectando sem haver a colisão real desejada. O dragão colide com uma área vazia do sprite do herói.

A abordagem que adotaremos para resolver este problema é a seguinte: cada sprite poderá ter **vários** retângulos de colisão, demarcando suas principais áreas:

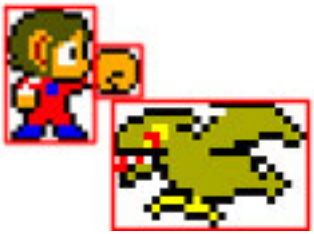


Figura 5.4: Definindo vários bounding boxes para objetos irregulares

Nosso controle agora ficará muito mais preciso e com qualidade bastante satisfatória para a maioria dos casos. Colisão em pequenas áreas vazias serão imperceptíveis devido à velocidade com que ocorrem as animações (e sempre poderemos definir mais um retângulo onde acharmos necessário).

Ganhamos também um bônus: na figura 5.4, temos uma ação de ataque (o herói tentando socar o dragão). Podemos diferenciar quem conseguiu atacar quem através de qual retângulo do herói colidiu com o retângulo do dragão.

Interseção de retângulos

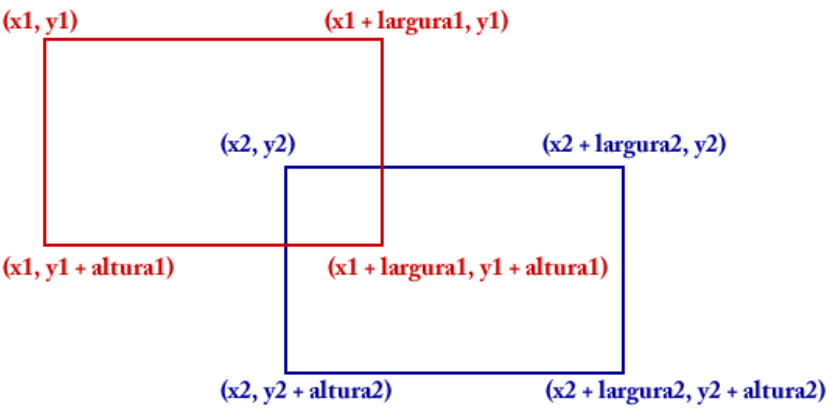


Figura 5.5: Intersecção de retângulos

Pela geometria, dois retângulos se intersectam se:

- (x1 + largura1) > x2 **E**
- x1 < (x2 + largura2) **E**
- (y1 + altura1) > y2 **E**
- y1 < (y2 + altura2)

No próximo tópico, iniciaremos o desenvolvimento de uma classe que aplica essas fórmulas para detectar colisões entre sprites.

5.2 Teste da classe Colisor

Iniciaremos o desenvolvimento da classe responsável por colidir objetos. Como de costume, iniciaremos por um pequeno aplicativo de teste onde definimos como queremos interagir com a classe.

Criaremos uma classe para essa nova funcionalidade. Começaremos criando o aplicativo de teste, onde definimos de que forma queremos interagir com a classe `Colisor`:

```
<!-- arquivo: teste-colisao-1.html -->
<!DOCTYPE html>
<html>

<head>
   <title>Detecção de colisões</title>
   <script src="colisor.js"></script>
   <script src="bola.js"></script>
</head>

<body>
   <canvas id="canvas_colisao" width="500" height="500">
   </canvas>
   <script>
      // As bolinhas, além de quicar, baterão umas nas outras
   </script>
</body>

</html>
```

No JavaScript, obtemos o canvas e seu *context*, e criamos algumas bolinhas:

```
var canvas = document.getElementById('canvas_colisao');
var context = canvas.getContext('2d');

var b1 = new Bola(context);
b1.x = 200;
b1.y = 200;
b1.velocidadeX = 10;
b1.velocidadeY = -5;
b1.cor = 'blue';
b1.raio = 20;

var b2 = new Bola(context);
```

```
b2.x = 300;
b2.y = 300;
b2.velocidadeX = -5;
b2.velocidadeY = 10;
b2.cor = 'red';
b2.raio = 30;
```

O próximo passo é criar o objeto `Colisor` e colocar nele os sprites:

```
var colisor = new Colisor();
colisor.novoSprite(b1);
colisor.novoSprite(b2);
```

Agora, façamos o loop de animação. Por enquanto, este loop será uma simples função. Perceba que, a cada ciclo, apenas realizamos as tarefas que nossos loops já fazem e, em seguida, mandamos o colisor realizar seu processamento. O colisor se encarregará de enviar mensagens aos objetos que colidirem.

```
requestAnimationFrame(animar);

function animar() {
   // Limpando a tela
   context.clearRect(0, 0, canvas.width, canvas.height);

   // Atualizando os sprites
   b1.atualizar();
   b2.atualizar();

   // Desenhando
   b1.desenhar();
   b2.desenhar();

   // Processar colisões
   colisor.processar();

   requestAnimationFrame(animar);
}
```

5.3 A classe Colisor

A classe `Colisor`, como definimos antes, terá inicialmente 2 métodos: `novoSprite` e `processar`. Manteremos um array de sprites, da mesma forma que a classe `Animacao`:

```
function Colisor() {
   this.sprites = [];
}
Colisor.prototype = {
   novoSprite: function(sprite) {
      this.sprites.push(sprite);
   },
   processar: function() {
      // Aqui faremos a verificação de colisão
   }
}
```

O coração da classe é o método `processar`. Aqui, precisamos testar quais sprites estão colidindo entre si. Faremos um algoritmo do tipo "todos contra todos" no array de sprites, usando dois loops `for` aninhados. Para cada elemento no array (primeiro loop), é testada sua colisão contra todos os outros (segundo loop). Note que ainda estamos abstraindo a lógica de cálculo da colisão:

```
processar: function() {
   for (var i in this.sprites) {
      for (var j in this.sprites) {
         // Não colidir um sprite com ele mesmo
         if (i == j) continue;

         // Abstrair a colisão
         this.testarColisao(this.sprites[i], this.sprites[j]);
      }
   }
}
```

O novo método `testarColisao` irá fazer mais um "todos contra todos", agora entre os retângulos do primeiro sprite e os do segundo. Lembre-se de que cada sprite poderá definir mais de um retângulo (figura 5.4). Dessa

forma, criamos um label `colisoes` para podermos parar os dois loops logo na primeira colisão entre retângulos detectada:

```
testarColisao: function(sprite1, sprite2) {
   // Obter os retângulos de colisão de cada sprite
   var rets1 = sprite1.retangulosColisao();
   var rets2 = sprite2.retangulosColisao();

   // Testar as colisões entre eles
   colisoes:
   for (var i in rets1) {
      for (var j in rets2) {
         // Ainda abstraindo a fórmula!
         if (this.retangulosColidem(rets1[i], rets2[j])) {
            // Eles colidem, vamos notificá-los
            sprite1.colidiuCom(sprite2);
            sprite2.colidiuCom(sprite1);

            // Não precisa terminar de ver todos os retângulos
            break colisoes;
         }
      }
   }
}
```

Esta parte é muito importante, pois definimos uma nova *interface* que os sprites deverão seguir para poderem participar do colisor. Essa interface é formada por dois métodos:

- `retangulosColisao`: deve devolver um array com os dados de cada retângulo;
- `colidiuCom(sprite)`: recebe a notificação de que o sprite colidiu com outro (recebido como parâmetro)

Nós abstraímos mais um pouco, deixando para o final a função que verifica se dois retângulos colidem. Considero uma excelente prática ir isolando as partes "difíceis" ou que não têm muito a ver com o algoritmo sendo criado:

```
retangulosColidem: function(ret1, ret2) {
   // Fórmula de interseção de retângulos
   return (ret1.x + ret1.largura) > ret2.x &&
          ret1.x < (ret2.x + ret2.largura) &&
          (ret1.y + ret1.altura) > ret2.y &&
          ret1.y < (ret2.y + ret2.altura);
}
```

5.4 Criando um sprite colidível

Agora que temos nosso `Colisor`, vamos criar uma nova versão da classe `Bola` seguindo a interface exigida por este, além da interface de animação já vista anteriormente:

```
function Bola(context) {
   this.context = context;
   this.x = 0;
   this.y = 0;
   this.velocidadeX = 0;
   this.velocidadeY = 0;
   this.cor = 'black';
   this.raio = 10;
}
Bola.prototype = {
   // Interface de animação
   atualizar: function() {

   },
   desenhar: function() {

   },
   // Interface de colisão
   retangulosColisao: function() {

   },
   colidiuCom: function(sprite) {

   }
}
```

Os métodos `atualizar` e `desenhar` não mudaram em nada. O método `atualizar` muda a posição da bola de acordo com suas velocidades horizontal e vertical e inverte as velocidades quando ela está quicando no Canvas. O método `desenhar` usa a API do contexto "2d" para desenhar a bola em sua posição, com a cor e o raio definidos. Em caso de dúvida, consulte a seção 2.4:

```
atualizar: function() {
   var ctx = this.context;

   if (this.x < this.raio || this.x >
         ctx.canvas.width - this.raio)
      this.velocidadeX *= -1;

   if (this.y < this.raio || this.y >
         ctx.canvas.height - this.raio)
      this.velocidadeY *= -1;

   this.x += this.velocidadeX;
   this.y += this.velocidadeY;
},
desenhar: function() {
   var ctx = this.context;
   ctx.save();
   ctx.fillStyle = this.cor;
   ctx.beginPath();
   ctx.arc(this.x, this.y, this.raio, 0, 2 * Math.PI, false);
   ctx.fill();
   ctx.restore();
}
```

Para a bola, o método `retangulosColisao`, criará um único retângulo que, mesmo assim, deve estar contido em um array. Só para lembrar, em JavaScript, os colchetes delimitam arrays e as chaves delimitam objetos. Temos, portanto, um objeto dentro de um array:

```
retangulosColisao: function() {
   return [
      {
```

```
        x: this.x - this.raio,  // this.x é o centro da bola
        y: this.y - this.raio,  // this.y idem
        largura: this.raio * 2,
        altura:  this.raio * 2
      }
   ];
},
```

Como a posição (x,y) da bola define seu centro, descontamos o raio para achar a posição do seu retângulo circundante, que na realidade é um quadrado. O lado desse quadrado é igual ao dobro do raio da bola:

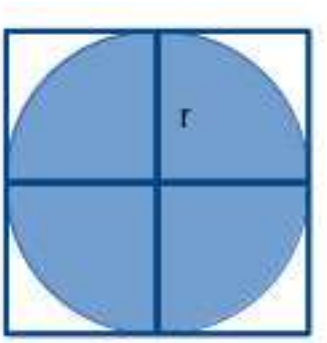


Figura 5.6: Retângulo circundante da bola

Falta agora o método `colidiuCom`. Por enquanto, vamos mostrar uma mensagem de alerta:

```
colidiuCom: function(sprite) {
   alert('PÁ!');
}
```

Faça o teste e veja se a mensagem aparece:

Figura 5.7: Colisão detectada

No entanto, a mensagem aparece muitas vezes pelos seguintes motivos:

- Estamos detectando colisões repetidas. Testamos A e vimos que colidiu com B. Ao testar B, não precisamos colidi-lo com A novamente!
- São duas bolinhas dando a mesma resposta à colisão, então tudo isso ocorre em dobro!
- Resultado: quatro "PÁ!" por colisão.
- Fora o fato de as bolinhas continuarem seu trajeto normalmente, fazendo com que passem uma "dentro" da outra e provoquem novas colisões.

5.5 Melhorando o código

Resolvendo a detecção repetida

Do jeito que estamos programando, cada colisão é detectada duas vezes. Fizemos a seguinte sequência de ações:

- Pegamos o primeiro sprite (vamos chamá-lo de **A**);

- Fazemos um loop em todos os outros e testamos contra A;
- Detectamos uma colisão com outro sprite (que chamaremos de **B**);
- Enviamos uma mensagem para A e para B.

Mas o processo não para aí, chegou a vez de testar B!

- Testamos B contra os outros;
- Detectamos sua colisão com A (que já foi detectada anteriormente);
- Enviamos as mensagens novamente para A e para B.

Para resolver o primeiro problema, vamos manter um registro de colisões já testadas. Isto pode ser feito de diversas formas, mas queimando neurônios eu achei mais fácil usar um objeto associativo contendo arrays, onde eu associo um sprite aos outros com quem ele já colidiu. Veja:

```
// Exemplo teórico

// Crio um objeto vazio
var jaTestados = new Object();

// Associo um sprite a um array de outros
// Estou dizendo que a nave já colidiu com o meteoro e o item
var jaTestados[nave] = [meteoro, item];
```

Infelizmente, os elementos de `jaTestados` não podem ser outros objetos. Nesse código, `nave` deve ser uma string, pois as propriedades de um objeto podem ser expressas por strings usando a sintaxe de *[colchetes]* (veja a seção 1.4).

Preste bastante atenção ao exemplo teórico a seguir. Se gerarmos uma string única para cada sprite, podemos associar cada uma a um array dentro desse objeto associativo. Tendo as strings devidamente armazenadas no objeto, podemos verificar se a colisão já foi testada:

```
// Exemplo teórico
// A cada ciclo do loop estamos testando 2 sprites

// Criar as identificações
var id1 = stringUnica(sprite1);
var id2 = stringUnica(sprite2);

// Criar os arrays se não existirem
if (! jaTestados[id1]) jaTestados[id1] = [];
if (! jaTestados[id2]) jaTestados[id2] = [];

// Testar a colisão se já não foi testado id1 com id2
testarSeNaoRepetido(id1, id2);

// Registrando o teste para evitar repetição
jaTestados[id1].push(id2);
jaTestados[id2].push(id1);
```

Para verificar se o teste atual não é repetido, podemos usar o método `indexOf` dos arrays. Por exemplo, `jaTestados[id1].indexOf(id2)` devolve a posição da string `id2` dentro do array associado à `id1` no objeto, ou o valor -1 caso ainda não exista:

```
// Exemplo teórico (parte final)
// Verificar se id1 com id2 já foi testado
if (! (jaTestados[id1].indexOf(id2) >= 0 ||
       jaTestados[id2].indexOf(id1) >= 0) ) {

   // Aqui testamos a colisão
}
```

Vamos, finalmente, para a parte prática. Primeiro, crie o método `stringUnica`, que gerará uma string para o sprite a partir dos dados de seus retângulos de colisão:

```
stringUnica: function(sprite) {
   var str = '';
   var retangulos = sprite.retangulosColisao();
```

```
    for (var i in retangulos) {
        str += 'x:' + retangulos[i].x + ',' +
               'y:' + retangulos[i].y + ',' +
               'l:' + retangulos[i].largura + ',' +
               'a:' + retangulos[i].altura + '\n';
    }

    return str;
}
```

Acompanhe a nova implementação do método `processar`, acrescentando os novos testes. Iniciamos com um objeto vazio e, a cada verificação, geramos uma string para cada sprite, verificamos se seus arrays já existem e se já foi testada essa colisão. Caso não, nós a testamos e a registramos:

```
processar: function() {
    // Inicio com um objeto vazio
    var jaTestados = new Object();

    for (var i in this.sprites) {
        for (var j in this.sprites) {
            // Não colidir um sprite com ele mesmo
            if (i == j) continue;

            // Gerar strings únicas para os objetos
            var id1 = this.stringUnica(this.sprites[i]);
            var id2 = this.stringUnica(this.sprites[j]);

            // Criar os arrays se não existirem
            if (! jaTestados[id1]) jaTestados[id1] = [];
            if (! jaTestados[id2]) jaTestados[id2] = [];

            // Teste de repetição
            if (! (jaTestados[id1].indexOf(id2) >= 0 ||
                   jaTestados[id2].indexOf(id1) >= 0) ) {

                // Abstrair a colisão
                this.testarColisao(this.sprites[i],
                                   this.sprites[j]);
```

```
            // Registrando o teste
            jaTestados[id1].push(id2);
            jaTestados[id2].push(id1);
        }
    }
  }
},
```

Se você rodar o programa agora, verá que reduzimos para dois "PÁ!" por colisão, pois são duas bolinhas respondendo.

Um tratador geral de colisões

Por que somente as bolinhas (sprites) podem responder à colisão? Há casos, por exemplo, em que eu quero uma resposta única e não é necessário que os dois objetos respondam. Que tal poder fazer:

```
// Exemplo teórico
colisor.aoColidir = function(sprite1, sprite2) {
  // Resposta única
};
```

O atributo `aoColidir` seria uma função de callback executada em toda e qualquer colisão. Esta função receberia como parâmetros os sprites que colidiram.

No construtor da classe `Colisor`, inicie o atributo `aoColidir` com null, somente para documentá-lo:

```
function Colisor() {
  this.sprites = [];
  this.aoColidir = null;
}
```

No método `testarColisao`, no ponto em que as mensagens são enviadas, acrescente a chamada a esse callback. Usamos um `if` para verificar se foi atribuída uma função:

```
//...
```

```
if (this.retangulosColidem(rets1[i], rets2[j])) {
   // Eles colidem, vamos notificá-los
   sprite1.colidiuCom(sprite2);
   sprite2.colidiuCom(sprite1);

   // Tratador geral
   if (this.aoColidir) this.aoColidir(sprite1, sprite2);

   // Não precisa terminar de ver todos os retângulos
   break colisoes;
}

//...
```

No aplicativo de teste, logo após a criação do colisor, podemos atribuir a seguinte função:

```
var colisor = new Colisor();
colisor.novoSprite(b1);
colisor.novoSprite(b2);

colisor.aoColidir = function(s1, s2) {
   alert('PÁ');
};
```

E o método `colidiuCom` da `Bola` ficará com o corpo vazio (por enquanto):

```
colidiuCom: function(sprite) {

}
```

Neste ponto, temos apenas um "PÁ!" para cada deslocamento. Lembre-se de que as bolinhas continuam sua trajetória em linha reta e provocam novas colisões enquanto passam uma pela outra.

Fazendo os sprites quicarem

Se fizermos as bolinhas quicarem uma na outra, resolvemos completamente o problema do excesso de colisões. O segredo, aqui, é verificar qual

bolinha está em que lado, e fazer com que passe a ir nesse sentido (não importa em que sentido esteja indo). A figura 5.8 representa isso na horizontal:

Figura 5.8: O sprite que está à esquerda vai para a esquerda, e o que está à direita vai para a direita

O mesmo vale para a vertical:

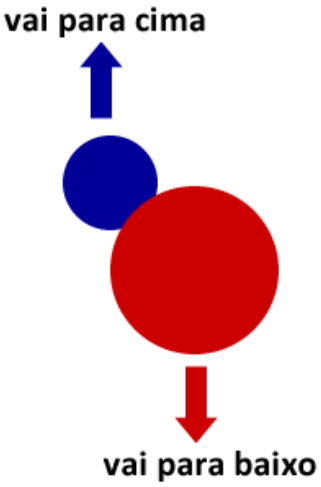


Figura 5.9: O sprite que está acima vai para cima, e o que está abaixo vai para baixo

Faremos isto no método `colidiuCom` da classe `Bola`. Para forçar a bola a ir para um determinado lado, obtemos os módulos das velocidades com `Math.abs` e colocamos um sinal negativo nos casos em que a velocidade passa a ser negativa (para esquerda e para cima):

```
colidiuCom: function(sprite) {
   if (this.x < sprite.x)  // Estou na esquerda
      this.velocidadeX = -Math.abs(this.velocidadeX);  // -
   else
      this.velocidadeX = Math.abs(this.velocidadeX);   // +

   if (this.y < sprite.y)  // Estou acima
      this.velocidadeY = -Math.abs(this.velocidadeY);  // -
   else
      this.velocidadeY = Math.abs(this.velocidadeY);   // +
}
```

Temos agora uma única mensagem "PÁ!" por colisão. Se você quiser ver apenas as bolinhas quicarem, basta comentar o `alert` ou simplesmente remover o tratador geral:

```
// Remova este código ou comente o corpo da função:
colisor.aoColidir = function(s1, s2) {
   //alert('PÁ');
};
```

Aqui, as possibilidades são infinitas. No decorrer do desenvolvimento de nosso jogo, faremos os objetos atingidos por tiros sumirem, criaremos novos sprites (explosões), aumentamos indicadores na tela (coleta de item) etc.

No próximo capítulo, faremos as primeiras experiências para o jogo de nave ilustrado no primeiro capítulo.

Capítulo 6

Iniciando o desenvolvimento do jogo

A partir deste ponto daremos início ao desenvolvimento de nosso jogo de nave. O *game engine* ainda terá muitos acréscimos e melhorias, conforme formos precisando de novos recursos. Em algum momento, ele ficará maduro e pronto para uso, mas minha ideia é mostrar como ele pode sempre continuar evoluindo conforme vamos desenvolvendo jogos com ele.

6.1 Animação de fundo com efeito parallax

A primeira coisa que faremos é o fundo. Se você ainda não fez o download dos arquivos deste livro, faça em http://github.com/EdyKnopfler/games-js/archive/master.zip, pois usaremos as imagens contidas ali.

O cenário do jogo terá o seguinte aspecto, em resolução de 500 por 500 pixels:

Figura 6.1: Cenário do jogo de nave

O fundo, no entanto, será composto por 3 imagens separadas: uma para o gradiente, outra para as estrelas e outra para as nuvens. Estas duas últimas terão fundo transparente, para podermos encaixar umas sobre as outras, como camadas:

Figura 6.2: Combinando os elementos do fundo

Cada imagem possui 500 pixels de largura por 1000 pixels de altura (portanto, o dobro da área do jogo). Elas rolarão pelo cenário *a velocidades diferentes*, criando um efeito denominado *parallax* (paralaxe):

Figura 6.3: Os elementos do fundo rolam a velocidades diferentes

Paralaxe

Um avião no céu parece voar a uma velocidade incrivelmente lenta. No entanto, se estivéssemos lá em cima e ele passasse por nós, o veríamos a toda velocidade.

Da mesma forma, viajando por uma estrada, o mato diante do meio-fio e as placas passam velozmente, enquanto a serra ao longe se afasta devagar.

Esse efeito de velocidade aparente, na Física, é denominado *paralaxe*. **Quanto mais distante está um objeto, menor será sua velocidade percebida por nós**.

O livro Core HTML5 Canvas, de David Geary, traz um exemplo muito bem feito, que pode ser conferido aqui:

http://corehtml5canvas.com/code-live/ch05/example-5.17/example.html

Obs.: entre desenvolvedores de jogos, o mais comum é nos referirmos ao efeito pelo nome em inglês (*parallax*), razão pela qual usarei esta forma daqui para frente.

Durante a rolagem, cada imagem deverá ser desenhada no mínimo duas vezes, de forma a cobrir toda a área de desenho durante a rolagem (figura 6.4). Se fosse uma imagem menor, teríamos que desenhá-la mais vezes. A cada ciclo, nós emendamos os desenhos a uma altura diferente (marcada com uma linha vermelha):

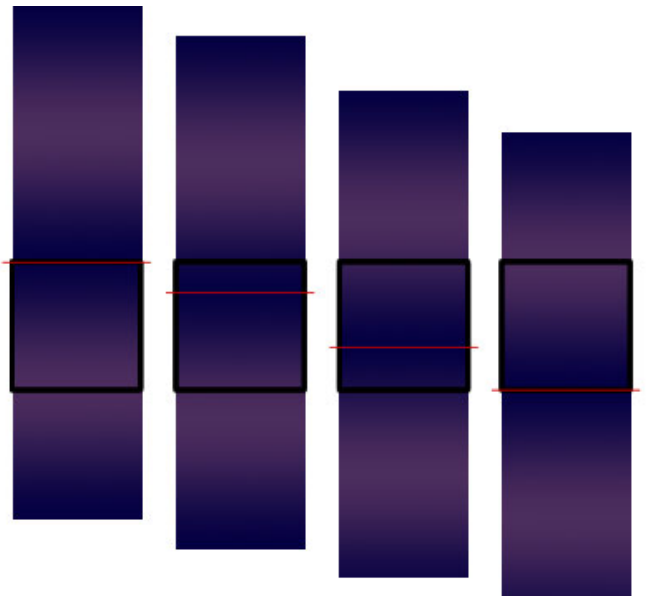


Figura 6.4: Cada imagem de fundo é desenhada quantas vezes forem necessárias para cobrir a área do jogo. Com imagens maiores que o Canvas, duas vezes são suficientes

Repare que os pontos inicial e final do gradiente têm de possuir a mesma cor para serem emendados (lembre-se disso ao criar as imagens em seu programa gráfico predileto). Da mesma forma, os extremos das outras figuras precisam encaixar-se perfeitamente! Por exemplo, para criar as nuvens, podemos proceder da maneira descrita nas figuras 6.5 e 6.6:

Figura 6.5: Quebramos uma nuvem pelo centro

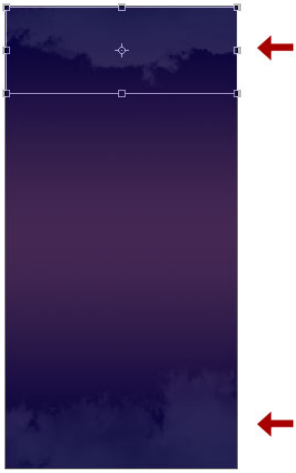


Figura 6.6: Viramos cada metade de cabeça para baixo e encaixamos nas bordas da imagem

Tomando esses cuidados ao produzir as imagens de fundo de seus games,

a programação ficará muito fácil. Vamos criar a classe `Fundo`, que será basicamente um sprite como todos os outros.

Teste para a classe `Fundo`

Para a criação deste teste, crie uma pasta de nome `img` na mesma pasta onde o HTML será salvo, e copie nela os arquivos `fundo-espaco.png`, `fundo-estrelas.png` e `fundo-nuvens.png` do pacote que você baixou. Você pode criar ou procurar outras imagens com fundo transparente também, mas certifique-se de recortá-las ou dimensioná-las para 500x1000.

Figura 6.7: Crie uma pasta junto ao HTML e copie as imagens nela

O HTML para o teste não tem nada de especial:

```
<!-- arquivo: teste-fundo.html -->
<!DOCTYPE html>
<html>

<head>
   <title>Fundos rolando em Parallax</title>
   <script src="animacao.js"></script>
   <script src="fundo.js"></script>
</head>
```

```
<body>
   <canvas id="canvas_fundo" width="500" height="500"></canvas>
   <script>
      // Primeiro temos que carregar as imagens
   </script>
</body>

</html>
```

Observação

Ao incluir arquivos já criados anteriormente (no caso, `animacao.js`), pegue sempre o do capítulo mais recente. Neste caso, a versão mais recente da classe `Animacao` é do capítulo 4. A cada capítulo ou tópico, o game engine poderá sofrer alterações.

No JavaScript, inicialmente mandamos carregar as imagens:

```
var imgEspaco = new Image();
imgEspaco.src = 'img/fundo-espaco.png';

var imgEstrelas = new Image();
imgEstrelas.src = 'img/fundo-estrelas.png';

var imgNuvens = new Image();
imgNuvens.src = 'img/fundo-nuvens.png';
```

Em seguida, temos que esperar todas serem carregadas. Como já aprendemos, isto é feito através do evento `onload`. Mantemos um contador de quantas já estão carregadas e, quando todas estiverem prontas, a aplicação pode de fato "iniciar":

```
var carregadas = 0;
var total = 3;

imgEspaco.onload = carregando;
imgEstrelas.onload = carregando;
```

```
imgNuvens.onload = carregando;

function carregando() {
   carregadas++;
   if (carregadas == total) iniciar();
}
```

Na função `iniciar`, fazemos as operações habituais: inicializamos o Canvas, criamos os sprites (que aqui serão os fundos rolando) e os incluímos no objeto animador:

```
function iniciar() {
   var canvas = document.getElementById('canvas_fundo');
   var context = canvas.getContext('2d');

   // Passo o context e a imagem para os objetos Fundo
   var fundoEspaco = new Fundo(context, imgEspaco);
   var fundoEstrelas = new Fundo(context, imgEstrelas);
   var fundoNuvens = new Fundo(context, imgNuvens);

   // Cada um a uma velocidade diferente
   fundoEspaco.velocidade = 3;
   fundoEstrelas.velocidade = 7;
   fundoNuvens.velocidade = 10;

   var animacao = new Animacao(context);
   animacao.novoSprite(fundoEspaco);
   animacao.novoSprite(fundoEstrelas);
   animacao.novoSprite(fundoNuvens);
   animacao.ligar();
}
```

Iniciando a classe `Fundo`

A próxima etapa é iniciar a classe `Fundo`. Sempre adotamos a prática de receber as imagens pelo construtor, para permitir que a aplicação controle quando todas estão carregadas:

```
// arquivo: fundo.js
function Fundo(context, imagem) {
```

```
    this.context = context;
    this.imagem = imagem;
    this.velocidade = 0;
}
Fundo.prototype = {
    atualizar: function() {

    },
    desenhar: function() {

    }
}
```

Lembre-se de que cada fundo deve ser desenhado duas vezes, emendados pelas extremidades em posições diferentes (figura 6.4). Crie o atributo `posicaoEmenda` e o inicialize no construtor com o valor zero:

```
function Fundo(context, imagem) {
    this.context = context;
    this.imagem = imagem;
    this.velocidade = 0;
    this.posicaoEmenda = 0;
}
```

No método `desenhar`, desenhamos as duas cópias da imagem emendadas nessa posição:

```
desenhar: function() {
    var img = this.imagem;  // Para facilitar a escrita :D

    // Primeira cópia
    var posicaoY = this.posicaoEmenda - img.height;
    this.context.drawImage(img, 0, posicaoY, img.width,
        img.height);

    // Segunda cópia
    posicaoY = this.posicaoEmenda;
    this.context.drawImage(img, 0, posicaoY, img.width,
        img.height);
}
```

No método `atualizar`, incrementamos a posição da emenda e a voltamos para zero caso a imagem de cima desça mais que o início do Canvas. A figura 6.8 adiante ilustra o que acontece:

```
atualizar: function() {
   // Atualizar a posição de emenda
   this.posicaoEmenda += this.velocidade;

   // Emenda passou da posição
   if (this.posicaoEmenda > this.imagem.height)
      this.posicaoEmenda = 0;
},
```

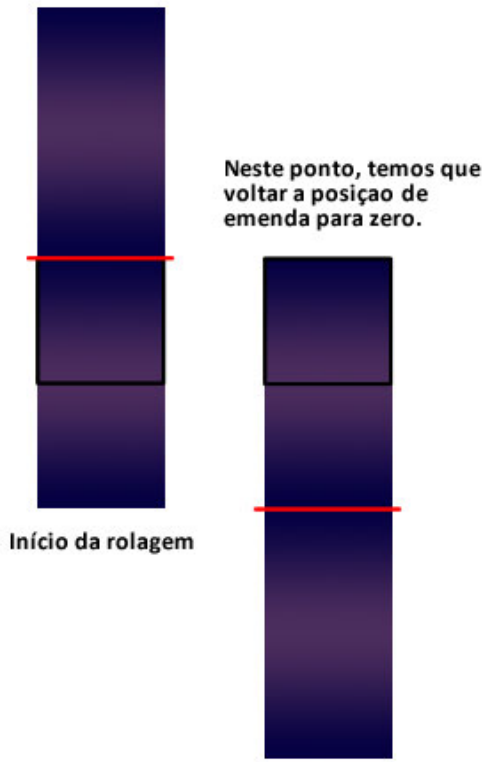


Figura 6.8: Imagens desceram muito, hora de reposicioná-las

Já estamos viajando pelo espaço sideral!

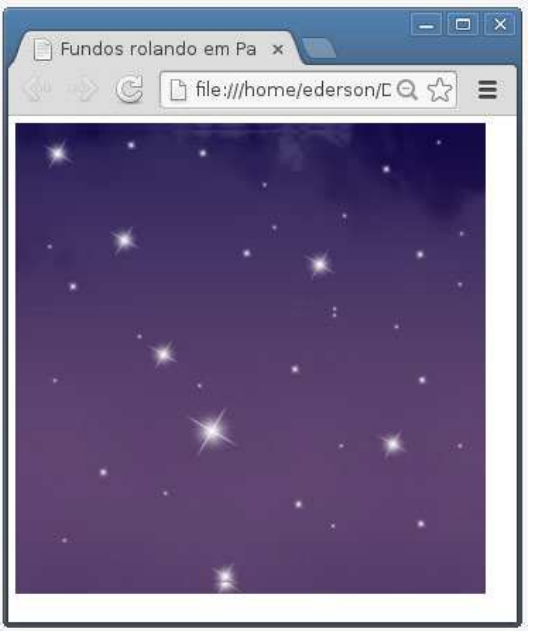


Figura 6.9: Imagens de fundo rolando em parallax

6.2 Controle da nave na horizontal e na vertical

Vamos criar uma nave controlável, inicialmente em um novo teste em branco, e depois juntando com o fundo em *parallax*. No pacote de download do livro, existe o arquivo `nave.png` (figura 6.10), com dimensões de 36 por 48 pixels. Você pode procurar ou criar outra imagem, mas certifique-se de que tenha fundo transparente.

Figura 6.10: Nave espacial controlável

Crie uma nova página para testarmos o controle da nave. Esta página usará as classes `Animacao`, `Teclado` e `Nave` (a ser criada):

```
<!-- arquivo: teste-nave.html -->
<!DOCTYPE html>
<html>

<head>
   <title>Nave Espacial Controlável</title>
   <script src="animacao.js"></script>
   <script src="teclado.js"></script>
   <script src="nave.js"></script>
</head>

<body>
   <canvas id="canvas_nave" width="500" height="500"></canvas>
   <script>
      // A nave andará por toda a tela!
   </script>
</body>

</html>
```

No JavaScript, temos o que já fizemos muitas vezes:

- referenciar o canvas e o context;
- instanciar as classes do game engine;
- instanciar o sprite (no caso, a nave) e sua imagem;
- aguardar a imagem carregar;

- iniciar a animação com a nave como sprite.

```
// Canvas e contexto
var canvas = document.getElementById('canvas_nave');
var context = canvas.getContext('2d');

// Teclado e animação (game engine)
var teclado = new Teclado(document);
var animacao = new Animacao(context);

// Sprite da nave e sua imagem
var imgNave = new Image();
imgNave.src = 'img/nave.png';
var nave = new Nave(context, teclado, imgNave);
animacao.novoSprite(nave);

// Quando carregar a imagem, iniciar a animação
imgNave.onload = function() {
   animacao.ligar();
}
```

A classe `Nave` receberá no construtor o context, o controlador do teclado e a imagem para desenho:

```
// arquivo: nave.js
function Nave(context, teclado, imagem) {
   this.context = context;
   this.teclado = teclado;
   this.imagem = imagem;
}
Nave.prototype = {
   atualizar: function() {

   },
   desenhar: function() {

   }
}
```

No método `atualizar`, lemos o estado de cada seta do teclado e move-

mos a nave de acordo. Para as setas à direita e abaixo, tivemos que considerar o tamanho do canvas e descontar o tamanho da nave, para que não ultrapassasse a borda:

```
atualizar: function() {
   if (this.teclado.pressionada(SETA_ESQUERDA) && this.x > 0)
      this.x -= this.velocidade;

   if (this.teclado.pressionada(SETA_DIREITA) &&
this.x < this.context.canvas.width - this.imagem.width)
      this.x += this.velocidade;

   if (this.teclado.pressionada(SETA_ACIMA) && this.y > 0)
      this.y -= this.velocidade;

   if (this.teclado.pressionada(SETA_ABAIXO) &&
this.y < this.context.canvas.height - this.imagem.height)
      this.y += this.velocidade;
}
```

> **Observação**
>
> Aqui usamos quatro `if`, sem o uso do `else`, para permitir que mais de uma seta possam estar pressionadas ao mesmo tempo. Isto nos permitirá mover a nave na diagonal!

A nossa necessidade criou três atributos: `x`, `y` e `velocidade`. Vamos iniciá-los no construtor:

```
function Nave(context, teclado, imagem) {
   this.context = context;
   this.teclado = teclado;
   this.imagem = imagem;
   this.x = 0;
   this.y = 0;
   this.velocidade = 0;
}
```

E setar valores adequados no código de teste, no evento `onload` da imagem, pois calcularemos a posição a partir das medidas da imagem e, portanto, ela deve estar carregada:

```
// Quando carregar a imagem, iniciar a animação
imgNave.onload = function() {
   // Centralizada na horizontal,
   // alinhada embaixo na vertical
   nave.x = canvas.width / 2 - imgNave.width / 2;
   nave.y = canvas.height - imgNave.height;
   nave.velocidade = 5;
   animacao.ligar();
}
```

No método `desenhar`, copiaremos a imagem da nave sem redimensioná-la:

```
desenhar: function() {
   this.context.drawImage(this.imagem, this.x, this.y,
         this.imagem.width, this.imagem.height);
}
```

Por último, certifique-se de que, no arquivo `teclado.js`, temos os códigos de todas as teclas de que precisamos:

```
// Códigos de teclas - aqui vão todos os que forem necessários
var SETA_ESQUERDA = 37;
var SETA_ACIMA = 38;
var SETA_DIREITA = 39;
var SETA_ABAIXO = 40;
var ESPACO = 32;
```

Voilà! Nossa nave move-se em todas as direções, sem ultrapassar os limites do Canvas!

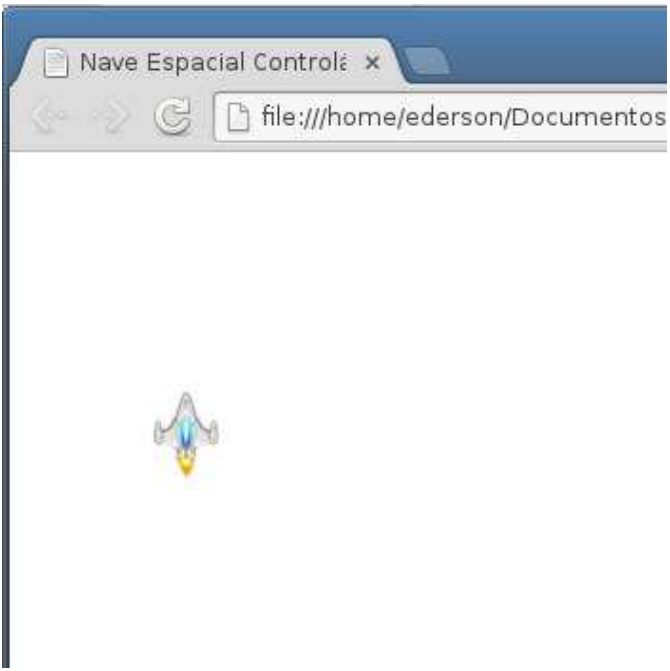


Figura 6.11: Controlando a nave pelas setas do teclado

6.3 Efetuando disparos

Agora a coisa vai começar a ficar divertida: nossa nave já se mexe, é hora de fazê-la disparar tiros!

Figura 6.12: Fazendo a nave atirar

Faça uma cópia do arquivo do teste anterior (`teste-nave.html`) e renomeie-a para `teste-tiros.html`. Logo antes do evento `onload` da imagem, coloque o código que manda a nave atirar ao disparar a tecla `Espaço`:

```
// Nunca foi tão fácil mandar uma nave atirar!
teclado.disparou(ESPACO, function() {
   nave.atirar();
});

// Quando carregar a imagem, iniciar a animação
imgNave.onload = function() {
   ...
}
```

Agora vamos criar o método `atirar` na classe `Nave`. Criar um tiro é fácil, o problema é incluí-lo na animação de sprites. Não temos nenhuma referência ao objeto `animacao`!

```
},
// Não esqueça da vírgula no último método quando criar outro
atirar: function() {
   var t = new Tiro(this.context);

   // Como incluir o novo tiro na animação?
}
```

Se você fez a lição de casa direitinho, poderia pensar: recebo pelo construtor! Sim, é uma grande ideia, e eu apoio. Receber as coisas pelo construtor é bem melhor do que fazer referência à variável `animacao` do aplicativo diretamente. E se eu tento reaproveitar esta classe onde não tenho esta variável? Receber pelo construtor forçaria o aplicativo a fornecer uma animação para a nave.

No entanto, pensando um pouco mais além, pode ser que eu tenha dezenas (ou centenas) de variedades de sprites que precisam criar outros na animação. Por que não damos a *todos* eles uma referência à `animacao`? Basta abrir a classe `Animacao` e modificar seu método `novoSprite`:

```
novoSprite: function(sprite) {
   this.sprites.push(sprite);
   sprite.animacao = this;
},
```

Agora qualquer sprite incluído tem uma referência ao objeto que controla a animação! Não é legal?

Podemos voltar ao método `atirar` da `Nave` e usar o objeto que ele recebeu:

```
atirar: function() {
   var t = new Tiro(this.context);
   this.animacao.novoSprite(t);
}
```

Enfim, vamos criar o esqueleto da classe `Tiro`:

```
// arquivo: tiro.js
function Tiro(context) {
```

```
    this.context = context;
}
Tiro.prototype = {
    atualizar: function() {

    },
    desenhar: function() {

    }
}
```

Onde o tiro vai aparecer? Na ponta da nave, você vai pensar. Dessa forma, é interessante recebermos a nave pelo construtor:

```
function Tiro(context, nave) {
    this.context = context;
    this.nave = nave;
}
```

No método `atirar`, a nave passa a si mesma para o tiro se posicionar:

```
atirar: function() {
    var t = new Tiro(this.context, this);
    this.animacao.novoSprite(t);
}
```

Tendo uma referência à nave, podemos posicionar o tiro. Faremos isso no próprio construtor:

```
function Tiro(context, nave) {
    this.context = context;
    this.nave = nave;

    // Posicionar o tiro no bico da nave
    this.largura = 4;
    this.altura = 20;
    this.x = nave.x + nave.imagem.width / 2 - this.largura / 2;
    this.y = nave.y - this.altura;
    this.velocidade = 10;
}
```

Para atualizar o tiro na animação, apenas o faremos subir na tela, subtraindo sua posição **y**:

```
atualizar: function() {
   this.y -= this.velocidade;
},
```

Para desenhá-lo, faremos um pequeno retângulo. Criamos um novo atributo, `cor`, para que possa ser alterado a gosto:

```
desenhar: function() {
   var ctx = this.context;
   ctx.save();
   ctx.fillStyle = this.cor;
   ctx.fillRect(this.x, this.y, this.largura, this.altura);
   ctx.restore();
}
```

No construtor, vamos inicializar `cor` com uma cor mais escura, pois a nave ainda está em fundo branco:

```
// Escolha sua cor!
this.cor = 'red';
```

Por último, não se esqueça de colocar a referência ao arquivo `tiro.js` na página HTML:

```
<head>
   <title>Nave Espacial Que Atira</title>
   <script src="animacao.js"></script>
   <script src="teclado.js"></script>
   <script src="nave.js"></script>
   <script src="tiro.js"></script>
</head>
```

Divirta-se! No próximo capítulo, iremos criar inimigos para o nosso herói.

Exercícios:

Se você entendeu bem os conceitos, as tarefas a seguir não deverão ser tão difíceis assim:

- Tente integrar o fundo em *parallax* com a nave se mexendo e atirando.
- Tente também atirar para os lados e para a diagonal. Isto requererá ler o estado das teclas de direção e dar velocidade horizontal ao tiro.

Comece a brincar e a experimentar!

Capítulo 7

Criando inimigos

Chegou a hora de adicionarmos um pouco de emoção verdadeira em nosso game. De nada adianta termos um jogo e não termos quem combater, não é mesmo? Portanto, criaremos agora os primeiros inimigos. Neste capítulo, utilizaremos bastante a classe `Colisor` para detectar quando os inimigos colidirem com os tiros e com a nave.

No pacote de arquivos do livro, está presente a imagem `ovni.png`. Nosso objetivo será fazer OVNIs caírem do alto da tela, em velocidades e posições x diferentes.

Figura 7.1: OVNIs cairão na tela!

7.1 Primeiro teste com nave e inimigos

Como de costume, vamos começar com a página contendo o aplicativo de teste. Veja como a quantidade de tags `<script>` está crescendo! Nosso protótipo está cada vez mais perto de um jogo 100% funcional.

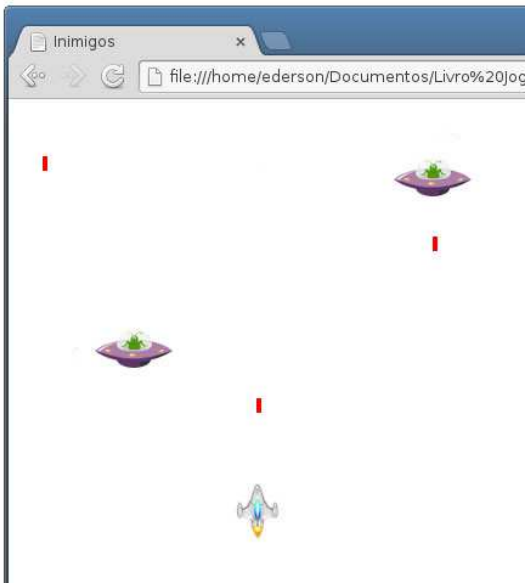


Figura 7.2: Cena com OVNIs e nave do herói

```
<!-- arquivo: teste-inimigos.html -->
<!DOCTYPE html>

<html>
```

```
<head>
   <title>Inimigos</title>
   <script src="animacao.js"></script>
   <script src="teclado.js"></script>
   <script src="nave.js"></script>
   <script src="tiro.js"></script>
   <script src="colisor.js"></script>
   <script src="ovni.js"></script>
</head>

<body>
   <canvas id="canvas_inimigos" width="500" height="500">
   </canvas>
   <script>
      // Inimigos terríveis tentarão destruí-lo!
   </script>
</body>

</html>
```

Para começar o JavaScript, vamos carregar as imagens, pois os sprites precisarão recebê-las pelos construtores. Como são duas, usaremos uma função `carregando` para detectar quando elas estiverem prontas:

```
// Imagens
var imgNave = new Image();
imgNave.src = 'img/nave.png';
imgNave.onload = carregando;

var imgOvni = new Image();
imgOvni.src = 'img/ovni.png';
imgOvni.onload = carregando;
```

Em seguida, a criação dos objetos básicos: canvas e contexto, teclado, animação, sprite da nave e colisor. Também configuramos o disparo pela tecla `Espaço`. Pode parecer repetitivo, mas a ideia é essa mesmo: tornar o negócio padronizado (e dessa forma você fixa melhor os conceitos)!

```
// Inicialização dos objetos
var canvas = document.getElementById('canvas_inimigos');
var context = canvas.getContext('2d');

var teclado = new Teclado(document);
var animacao = new Animacao(context);

var nave = new Nave(context, teclado, imgNave);
animacao.novoSprite(nave);

var colisor = new Colisor();
colisor.novoSprite(nave);

teclado.disparou(ESPACO, function() {
   nave.atirar();
});
```

Agora crie as funções `carregando`, que monitora o carregamento das imagens, e `iniciar`, chamada por `carregando` quando todas as imagens estiverem prontas. Na função `iniciar`, posicionamos a nave, iniciamos a animação e definimos que a cada 1000 milissegundos um novo inimigo será criado na tela, usando a função `setInterval` do JavaScript:

```
// Carregamento e inicialização
var carregadas = 0;

function carregando() {
   carregadas++;
   if (carregadas == 2) iniciar();
}

function iniciar() {
   nave.x = canvas.width / 2 - imgNave.width / 2;
   nave.y = canvas.height - imgNave.height;
   nave.velocidade = 5;
   animacao.ligar();

   setInterval(novoOvni, 1000);
}
```

Quem criará o inimigo é a função `novoOvni`. Sabendo que a imagem possui 64x32 pixels, nós a posicionamos acima da área visível do jogo, descontando 32 pixels de altura. Para dar um pouco de emoção ao game, a posição horizontal e a velocidade são definidas aleatoriamente para cada novo inimigo, com o auxílio de `Math.random` e `Math.floor`. Também é preciso colocar o inimigo no colisor:

```
// Criação dos inimigos
function novoOvni() {
   var ovni = new Ovni(context, imgOvni);

   // Mínimo: 5; máximo: 20
   ovni.velocidade =
      Math.floor( 5 + Math.random() * (20 - 5 + 1) );

   // Mínimo: 0;
   // máximo: largura do canvas - largura do ovni
   ovni.x =
      Math.floor(Math.random() *
                 (canvas.width - imgOvni.width + 1) );

   // Descontar a altura
   ovni.y = -imgOvni.height;

   animacao.novoSprite(ovni);
   colisor.novoSprite(ovni);
}
```

Combinando Math.random e Math.floor

O método `random` retorna um número aleatório fracionário, entre o e 1. Para gerar um número aleatório inteiro, usamos a fórmula:

```
function aleatorio(min, max) {
    return min + Math.floor(Math.random() * (max - min + 1));
}
```

O método `floor` descarta a parte fracionária do resultado, arredondando-o para o inteiro de menor valor (não o mais próximo).

Padronizando a codificação

Vamos sempre adotar a seguinte sequência de codificação ao criar novos aplicativos:

- Primeiro, carregamos as imagens, pois os objetos do jogo dependem delas;
- Em seguida, instanciamos os objetos do game engine (animação, teclado, colisor) e os sprites, usando as imagens quando necessário;
- Por último, criamos as funções de inicialização, que só devem executar quando as imagens estiverem completamente carregadas.

7.2 A classe Ovni

Vamos criar a classe `Ovni`, inicialmente sem se preocupar com a interface de colisão. Como de costume, começamos construindo um esqueleto a partir do que idealizamos para a classe na página HTML:

```
// arquivo: ovni.js
function Ovni(context, imagem) {
   this.context = context;
   this.imagem = imagem;
   this.x = 0;
   this.y = 0;
   this.velocidade = 0;
}
Ovni.prototype = {
   atualizar: function() {

   },
   desenhar: function() {

   }
}
```

Se o objetivo é fazer o inimigo descer pela tela, o método `atualizar` tem de incrementar a posição *y*:

```
atualizar: function() {
   this.y += this.velocidade;
},
```

Já o método `desenhar` apenas desenha a imagem que foi recebida no construtor:

```
desenhar: function() {
   var ctx = this.context;
   var img = this.imagem;
   ctx.drawImage(img, this.x, this.y, img.width, img.height);
}
```

Já temos uma nave que atira (programada no capítulo anterior) e OVNIs descendo no mesmo aplicativo (figura 7.2)!

7.3 Adicionando fundo em parallax

No capítulo anterior, criamos um fundo consistindo de várias imagens em movimento, cada uma em velocidade diferente, criando um efeito que

chamamos *parallax*. Chegou a hora de integrar esse fundo ao nosso game.

Primeiro, faça uma cópia do arquivo `teste-inimigos.html`, dando-lhe o nome `teste-inimigos-e-fundo.html`. Iremos fazer algumas alterações em determinados pontos.

No início do código, onde carregamos as imagens, adicione o carregamento das três imagens de fundo:

```
// Imagens
var imgEspaco = new Image();
imgEspaco.src = 'img/fundo-espaco.png';
imgEspaco.onload = carregando;

var imgEstrelas = new Image();
imgEstrelas.src = 'img/fundo-estrelas.png';
imgEstrelas.onload = carregando;

var imgNuvens = new Image();
imgNuvens.src = 'img/fundo-nuvens.png';
imgNuvens.onload = carregando;

// Imagens da nave e dos ovnis...
```

Agora temos cinco imagens no total, portanto temos que alterar a função `carregando` para contar até 5. Uma variável `total` nos permitirá alterar este parâmetro mais facilmente:

```
// Carregamento e inicialização
var carregadas = 0;
var total = 5;

function carregando() {
   carregadas++;
   if (carregadas == total) iniciar();
}
```

Precisamos adicionar os fundos à animação *antes* dos outros sprites, de forma que eles sejam desenhados primeiro. Na seção onde inicializamos os objetos do game engine e os sprites, **antes da criação da nave**, adicione o código:

```
// Primeiro os objetos do game engine
var canvas = ...
var context = ...
var teclado = ...
var animacao = ...

// Em seguida as imagens de fundo
var fundo1 = new Fundo(context, imgEspaco);
fundo1.velocidade = 3;
animacao.novoSprite(fundo1);

var fundo2 = new Fundo(context, imgEstrelas);
fundo2.velocidade = 7;
animacao.novoSprite(fundo2);

var fundo3 = new Fundo(context, imgNuvens);
fundo3.velocidade = 10;
animacao.novoSprite(fundo3);

// Depois do fundo, a nave e outros sprites
var nave = ...
```

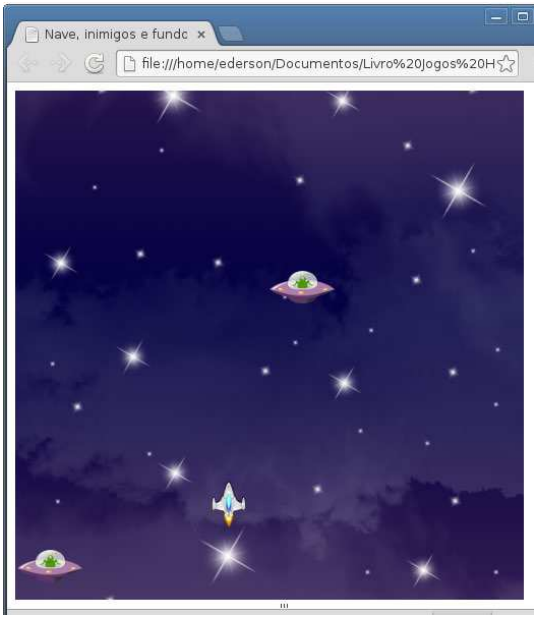


Figura 7.3: Cena com OVNIs, nave do herói e fundo rolando em parallax

Neste ponto, o fundo já se movimenta juntamente com a nave e os inimigos (figura 7.3). A partir de agora, como trabalharemos sempre com um fundo rolando, não é preciso mais realizar a limpeza de tela a cada ciclo da animação, bastando apenas desenhar o fundo para cobrir a tela. Isso não trará efeitos práticos mas economizará algum processamento. Por isso, vamos comentar a linha que limpa a tela no método `proximoFrame` da classe `Animacao`.

```
proximoFrame: function() {
    // Posso continuar?
    if ( ! this.ligado ) return;

    // Comente o comando que limpa a tela
    // this.limparTela();

    // ...
}
```

Obs.: recomendo que você faça uma cópia do arquivo `animacao.js`

especialmente para este tópico. Do contrário, testes anteriores serão afetados (figura 7.4). Para os próximos capítulos, você poderá usar a nova versão desse arquivo.

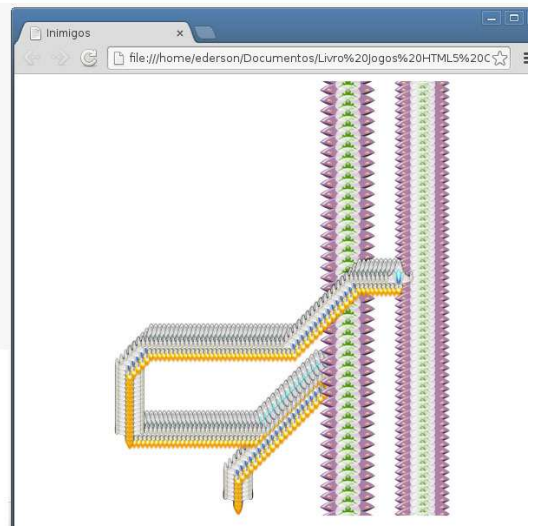


Figura 7.4: Mantenha a limpeza de tela para os testes anteriores, caso contrário serão prejudicados.

Experimente rodar o aplicativo. Já temos um protótipo de jogo quase pronto!

7.4 Adicionando colisão

A parte que falta para termos um protótipo funcional é detectar a colisão entre os sprites — mais exatamente, entre o inimigo e a nave (causando a morte do herói) e entre os inimigos e o tiro (causando a destruição do inimigo).

Precisamos, em algum ponto de nosso programa, chamar o método `processar` do `Colisor`. Que tal se a `Animacao` chamar o colisor em cada ciclo? Assim, teremos um controle mais geral, que ocorre em conjunto com o processamento dos ciclos de animação.

Indo mais além, a `Animacao` poderia manter uma lista de processamentos a executar, dos quais um é o `Colisor`. Mais adiante, podemos adicionar `Gravidade`, `TelaMovel` e por aí vai. Façamos isto!

No construtor da classe `Animacao`, adicione o comando que inicia um array vazio `processamentos`:

```
this.processamentos = [];
```

Crie o método `novoProcessamento`, para inserir novos processamentos no array:

```
novoProcessamento: function(processamento) {
   this.processamentos.push(processamento);
   processamento.animacao = this;
}
```

No método `proximoFrame`, vamos executar estes processamentos:

```
proximoFrame: function() {
   // Posso continuar?
   if ( ! this.ligado ) return;

   // Atualizamos o estado dos sprites
   for (var i in this.sprites)
      this.sprites[i].atualizar();

   // Desenhamos os sprites
   for (var i in this.sprites)
      this.sprites[i].desenhar();

   // Processamentos gerais
   for (var i in this.processamentos)
      this.processamentos[i].processar();

   // Chamamos o próximo ciclo
   var animacao = this;
   requestAnimationFrame(function() {
      animacao.proximoFrame();
   });
},
```

Iniciando o teste

Já temos o arquivo `teste-inimigos-e-fundo.html`, com tudo o que foi feito até aqui. Faça uma cópia e dê o nome `teste-colisao-inimigos.html`. Localize o ponto onde o colisor é criado e acrescente-o como um processamento da animação:

```
var colisor = new Colisor();
colisor.novoSprite(nave);
animacao.novoProcessamento(colisor);
```

Os inimigos já estão sendo adicionados no colisor, porém faltam os tiros. Eles são criados no método `atirar` da `Nave`. Modifique-o para que o tiro criado seja incluído no colisor:

```
atirar: function() {
   var t = new Tiro(this.context, this);
   this.animacao.novoSprite(t);
   this.colisor.novoSprite(t);
}
```

Mas... a nave tem referência ao colisor? Seria bom que cada sprite adicionado no colisor tivesse uma referência a este (assim como a `Animacao` já faz). Modifique o método `novoSprite` do `Colisor` para fazer isso:

```
novoSprite: function(sprite) {
   this.sprites.push(sprite);
   sprite.colisor = this;
},
```

Referências cruzadas

Vamos adotar a prática de, em qualquer objeto que agregue e processe vários outros objetos (`Animacao`, `Colisor` etc.), colocar neles uma referência ao objeto agregador. Por exemplo:

```
// Adiciono um sprite ao colisor
colisor.novoSprite(carro);

// O colisor adiciona-se a si mesmo no sprite
novoSprite: function(sprite) {
   this.sprites.push(sprite);
   sprite.colisor = this;
},
```

Adaptando os sprites à interface do colisor

Neste ponto, se tentarmos rodar o aplicativo, a animação irá parar abruptamente e teremos a seguinte mensagem no console (`Ctrl+Shift+J` no Google Chrome, `Ctrl+Shift+K` no Firefox):

Figura 7.5: Tentando chamar um método que não existe no objeto

Isso ocorreu porque agora o colisor está sendo chamado, e este, por sua vez, chama os métodos `retanguloColisao` e `colidiuCom`. Precisamos implementá-los nas classes `Nave`, `Tiro`, e `Ovni`. Faça isto nas três, por enquanto deixando os corpos dos métodos vazios:

```
retangulosColisao: function() {
```

```
},
colidiuCom: function(outro) {

}
```

Agora o aplicativo voltou a funcionar como a versão anterior. Vamos definir os retângulos de colisão dos sprites. O mais fácil é o do `Tiro`, que já é retangular:

```
retangulosColisao: function() {
   return [ {x: this.x, y: this.y, largura: this.largura,
         altura: this.altura} ];
},
```

Para a nave, vamos fazer três retângulos, para o corpo e as duas asas (figura 7.6). Não há uma regra fixa de quantos retângulos usar, você define a partir do formato de seu objeto. No caso de objetos mais complexos, é interessante desenhar os retângulos na tela, a fim de podermos ajustar os valores das posições mais facilmente:

```
retangulosColisao: function() {
   // Estes valores vão sendo ajustados aos poucos
   var rets =
   [
      {x: this.x+2, y: this.y+19, largura: 9, altura: 13},
      {x: this.x+13, y: this.y+3, largura: 10, altura: 33},
      {x: this.x+25, y: this.y+19, largura: 9, altura: 13}
   ];

   // Desenhando os retângulos para visualização
   var ctx = this.context;

   for (var i in rets) {
      ctx.save();
      ctx.strokeStyle = 'yellow';
      ctx.strokeRect(rets[i].x, rets[i].y, rets[i].largura,
                     rets[i].altura);
      ctx.restore();
   }
```

```
    return rets;
},
```

Figura 7.6: Retângulos de colisão da nave

Para analisar os retângulos de sprites em movimento, facilita muito abrir o console (`Ctrl+Shift+J` no Chrome; no Firefox instale o plugin Firebug) e chamar o comando: `animacao.desligar()`:

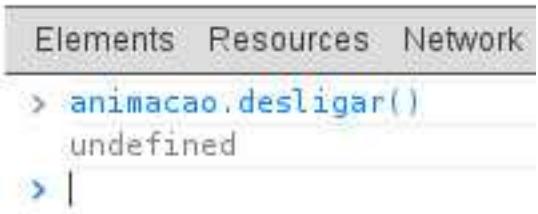


Figura 7.7: Pare a animação pelo console para analisar os retângulos de colisão

Para o OVNI, também usei três retângulos:

```
retangulosColisao: function() {
    var rets =
    [
        {x: this.x+20, y: this.y+1, largura: 25, altura: 10},
        {x: this.x+2, y: this.y+11, largura: 60, altura: 12},
        {x: this.x+20, y: this.y+23, largura: 25, altura: 7},
    ];
```

```
    // Desenho dos retângulos ...
}
```

Figura 7.8: Retângulos de colisão do OVNI

Rode o aplicativo e dê alguns tiros. Você rapidamente perceberá um problema...

7.5 Estamos experimentando lentidão!

Nosso jogo chegou em um ponto em que muitos objetos estão sendo criados, **mas não estão sendo destruídos**! Agora, além do loop de animação, eles também devem responder às chamadas do colisor. Isso, claro, está sobrecarregando o aplicativo.

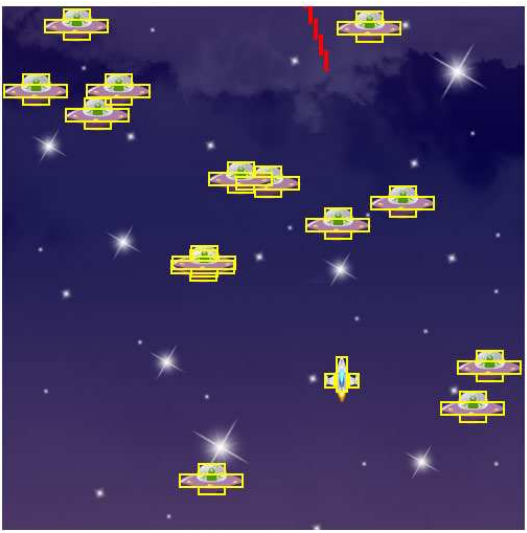


Figura 7.9: Experimente dar muitos tiros: nosso jogo ficará bem lento!

Nosso game engine, embora tenha funcionado perfeitamente até aqui, carece de algo bem básico: não é possível *excluir* objetos, tanto da animação quanto do colisor. Estamos sempre criando novos tiros e inimigos, mas não os destruímos quando estes deixam a tela. Resultado: continuam sendo processados.

O que precisamos para excluir tiros e inimigos que já se foram da tela?

Importante: siga este tópico com bastante atenção, pois faremos alterações delicadas no game engine. Procure **sempre** fazer backups de todos os arquivos.

Problema na exclusão de elementos

Antes de criar o método `excluirSprite` nas classes `Animacao` e `Colisor`, preciso contar-lhe uma coisa. Quando estava desenvolvendo o protótipo de jogo para este livro, ao implementar a exclusão de itens dos arrays, eu inocentemente mandei excluir o elemento tão logo que ele saía da tela:

```
// Exemplo teórico
```

```
// Não faça isto!

atualizar: function() {
   // Sprite sumiu da tela
   if (this.y > this.context.canvas.height) {
      this.animacao.excluirSprite(this);
      this.colisor.excluirSprite(this);
   }
}
```

É um código simples e claro, mas que provocou um bug que levei *horas* para descobrir o que era. Normalmente eu tenho deixado os bugs aparecerem para discutir a solução no próprio caminhar do livro, mas desta vez não vou fazer isso com você. Vou poupá-lo deste aqui.

A explicação é que estamos excluindo elementos de um array *dentro do loop que percorre esse array*! Mas onde está esse loop? Na classe `Animacao`, método `proximoFrame`, temos *vários* loops que chamam os métodos `atualizar`, `desenhar` e o `processar` do colisor, sendo que este faz outro loop em seu array!

```
// Trecho do método proximoFrame da classe Animacao

// Atualizamos o estado dos sprites
for (var i in this.sprites)
   this.sprites[i].atualizar();

// Desenhamos os sprites
for (var i in this.sprites)
   this.sprites[i].desenhar();

// Processamentos gerais
for (var i in this.processamentos)
   this.processamentos[i].processar();
```

Se, por exemplo, eu mando excluir um item de dentro do método `atualizar`, o loop que chama o `atualizar` ficará órfão daquele item! Como regra geral, você não deve excluir itens de um array dentro do loop que o percorre:

```
// Exemplo teórico
// Você não deve fazer isto

// Criar um array
var meuArray = ['banana', 'laranja', 'limão'];

// Fazer um loop
for (var i in meuArray) {
   // Forçar uma exclusão dentro do loop
   if (meuArray[i] == 'laranja')
      excluirElemento(meuArray[i], meuArray);

   // Fazer alguma tarefa com o array
   document.write(meuArray[i] + '<br>');
}
```

A solução? Manter uma lista de objetos a excluir, e excluí-los somente ao fim do ciclo de animação!

Excluindo elementos dos arrays

Vamos então implementar a exclusão de elementos com bastante calma. Desta forma, eliminamos o problema da lentidão e poderemos, enfim, tratar as colisões, fazendo o inimigo e o tiro sumirem quando se chocarem, por exemplo.

Temos que iniciar as listas de elementos a serem excluídos. No construtor da classe `Colisor`, faça:

```
this.spritesExcluir = [];
```

Para a classe `Animacao`, há dois arrays:

```
this.spritesExcluir = [];
this.processamentosExcluir = [];
```

O método `excluirSprite` apenas incluirá o sprite nessa lista. No `Colisor`, temos:

```
excluirSprite: function(sprite) {
   this.spritesExcluir.push(sprite);
}
```

Na `Animacao`, temos que ter `excluirSprite` e `excluirProcessamento`:

```
excluirSprite: function(sprite) {
   this.spritesExcluir.push(sprite);
},
excluirProcessamento: function(processamento) {
   this.processamentosExcluir.push(processamento);
}
```

No fim do método `processar` do `Colisor`, colocamos uma chamada a `processarExclusoes`:

```
processar: function() {
   // ...

   this.processarExclusoes();
},
```

E vamos criar esse método `processarExclusoes`. Como pode haver vários sprites a serem eliminados, vamos na verdade montar um novo array contendo todos os elementos, menos aqueles que foram excluídos. O array antigo poderá ser descartado e ficará livre para o coletor de lixo (*garbage collector*) do JavaScript apagá-lo da memória:

```
processarExclusoes: function() {
   // Criar um novo array
   var novoArray = [];

   // Adicionar somente os elementos não excluídos
   for (var i in this.sprites) {
      if (this.spritesExcluir.indexOf(this.sprites[i]) == -1)
         novoArray.push(this.sprites[i]);
   }

   // Limpar o array de exclusões
   this.spritesExcluir = [];

   // Substituir o array velho pelo novo
```

```
    this.sprites = novoArray;
}
```

Para a `Animacao`, o procedimento é muito semelhante. No fim do método `proximoFrame`, logo antes da chamada do próximo ciclo, chame também o método `processarExclusoes`:

```
proximoFrame: function() {
    // ...

    // Processamento de exclusões
    this.processarExclusoes();

    // Chamamos o próximo ciclo
    var animacao = this;
    requestAnimationFrame(function() {
        animacao.proximoFrame();
    });
},
```

E vamos criar esse método. Ele segue o mesmo algoritmo de seu equivalente no `Colisor`, só é um pouco maior porque estamos lidando com dois arrays.

```
processarExclusoes: function() {
    // Criar novos arrays
    var novoSprites = [];
    var novoProcessamentos = [];

    // Adicionar somente se não constar no array de excluídos
    for (var i in this.sprites) {
        if (this.spritesExcluir.indexOf(this.sprites[i]) == -1)
            novoSprites.push(this.sprites[i]);
    }

    for (var i in this.processamentos) {
        if (this.processamentosExcluir.indexOf
            (this.processamentos[i]) == -1)
            novoProcessamentos.push(this.processamentos[i]);
```

```
    }

    // Limpar os arrays de exclusões
    this.spritesExcluir = [];
    this.processamentosExcluir = [];

    // Substituir os arrays velhos pelos novos
    this.sprites = novoSprites;
    this.processamentos = novoProcessamentos;
}
```

A exclusão está implementada! Esta parte foi fogo, não foi? Mas agora podemos facilmente mandar excluir objetos que estão sobrando na memória, para que eles não provoquem mais lentidão!

7.6 Excluindo os objetos desnecessários

Vamos começar pelo `Tiro`. Quando sumir da tela, ele deve ser excluído. Em seu método `atualizar`, verificamos sua posição *y* e o tiramos da animação e do colisor quando sua posição for negativa (excedendo a borda do Canvas):

```
atualizar: function() {
    this.y -= this.velocidade;

    // Excluir o tiro quando sumir da tela
    if (this.y < -this.altura) {
        this.animacao.excluirSprite(this);
        this.colisor.excluirSprite(this);
    }
},
```

O `Ovni` também deve ser excluído, mas como seu movimento é em sentido contrário, ele deve passar da altura do Canvas:

```
atualizar: function() {
    this.y += this.velocidade;

    if (this.y > this.context.canvas.height) {
        this.animacao.excluirSprite(this);
```

```
        this.colisor.excluirSprite(this);
    }
},
```

Também, tanto o `Tiro` quanto o `Ovni` devem ser excluídos em caso de colisão entre eles. Afinal, esta é uma regra de negócio óbvia de nosso jogo! Podemos implementar isto no método `colidiuCom` de qualquer uma destas classes. Escolhi a classe `Ovni`, conferindo se o outro objeto é um `Tiro`:

```
colidiuCom: function(outro) {
    // Se colidiu com um Tiro, os dois desaparecem
    if (outro instanceof Tiro) {
        this.animacao.excluirSprite(this);
        this.colisor.excluirSprite(this);
        this.animacao.excluirSprite(outro);
        this.colisor.excluirSprite(outro);
    }
}
```

Por último, a colisão entre a `Nave` e o `Ovni`. Nosso herói, que era invencível, agora pode morrer! Programe na classe `Nave`:

```
colidiuCom: function(outro) {
    // Se colidiu com um Ovni...
    if (outro instanceof Ovni) {
        // Fim de jogo!
        this.animacao.desligar();
        alert('GAME OVER');
    }
}
```

Este é um momento para se comemorar! Se você fez todas as tarefas até aqui com calma e atenção, você tem agora um game engine maduro para uso e um exemplo de jogo totalmente funcional, embora ainda não esteja pronto.

Experimente jogar. Já é possível destruir os inimigos e ser atingido por eles. No próximo capítulo, realizaremos este e outros diversos ajustes que deixarão nosso jogo com aspecto profisisonal.

Exercício: tente fazer a nave sofrer 3 colisões antes de ser destruída. Para isso, basta criar um atributo (sugestões de nome: `energia`, `pontosVida`)

e ir decrementando seu valor a cada colisão. Quando chegar a zero... *GAME OVER*!

Capítulo 8

Incorpore animações, sons, pausa e vidas extras ao jogo

Uma boa notícia: neste capítulo, nosso jogo de nave será concluído! Já temos um protótipo totalmente funcional, mas ainda há muito o que ser melhorado para dar-lhe um aspecto profissional. Serão feitas as seguintes melhorias:

- **Organização do código:** nosso jogo está crescendo e, por isso, a página inicial vai começar a ficar um pouco longa e desorganizada. Faremos algumas refatorações para acomodar mais facilmente as mudanças que estão por vir.
- **Animação cronometrada:** para a animação correr a velocidades constantes, e não no ritmo da CPU (do jeito que está, a velocidade da animação poderá oscilar muito).

- **Uso de spritesheets:** nós aprendemos como usá-las no capítulo 4, mas não aplicamos ainda em nosso jogo! Com elas, criaremos explosões e daremos animação à nave, deixando nosso jogo muito mais interessante.
- **Pausa:** como seria a experiência de jogar um jogo sem poder pausá-lo? Péssima, não é mesmo?
- **Som e música de fundo:** você não estava sentindo a falta disto, não?
- **Telas de loading e Game Over:** um jogo utiliza muitas imagens. Quando você hospedá-lo na internet, não vai querer que o usuário veja uma tela parada enquanto as imagens carregam, não é verdade? Um aviso "Carregando..." e uma barrinha aumentando na tela farão com que o internauta não desista do seu jogo; ao contrário, aumentará sua expectativa! E uma tela que indica quando o jogo acabou, cairá muito bem.
- **Vidas extras:** por que o jogo tem que acabar na primeira colisão da nave? Vamos dar mais chances ao jogador!
- **Pontuação (score):** esta simples regra de negócio será bem fácil de implementar.

Primeiro, vamos organizar todo o nosso trabalho. Será uma excelente oportunidade de revisão. Mãos à obra!

8.1 Organizando o código

No capítulo anterior, o código da página HTML de nosso jogo ficou um tanto extenso, pois juntamos quase tudo o que aprendemos em um único arquivo funcional. Considero esta uma boa hora para refatorar esse código (e praticar um pouco também), deixando-o um pouco mais organizado e fácil para acrescentar as melhorias que estão por vir.

Crie uma nova página HTML. Os arquivos de script devem ser copiados de suas versões mais atualizadas.

```
<!-- arquivo: jogo-definitivo.html -->
<!DOCTYPE html>
<html>

<head>
   <title>Jogo de Nave</title>
   <script src="animacao.js"></script>
   <script src="teclado.js"></script>
   <script src="colisor.js"></script>
   <script src="fundo.js"></script>
   <script src="nave.js"></script>
   <script src="ovni.js"></script>
   <script src="tiro.js"></script>
</head>

<body>
   <canvas id="canvas_animacao" width="500" height="500">
   </canvas>
   <script>
      // Vamos organizar um pouco este aplicativo!
   </script>
</body>

</html>
```

Vamos continuar executando as tarefas nesta ordem: carregar as imagens, iniciar os objetos, iniciar a animação. Que tal criar funções separadas? No JavaScript, faça:

```
// Canvas e Context
var canvas = document.getElementById('canvas_animacao');
var context = canvas.getContext('2d');

// Variáveis principais
var imagens, animacao, teclado, colisor,
      nave, criadorInimigos;
var totalImagens = 0, carregadas = 0;

// Começa carregando as imagens
```

```
carregarImagens();
```

Na função `carregarImagens`, facilita muito carregar um objeto ou array com os nomes das imagens e fazer um loop nele. Optei por usar um objeto, pois assim posso associar cada imagem a seu nome (em vez de usar um número sem significado). Após carregar cada imagem, o nome é substituído pelo objeto da imagem:

```
function carregarImagens() {
   // Objeto contendo os nomes das imagens
   imagens = {
      espaco:   'fundo-espaco.png',
      estrelas: 'fundo-estrelas.png',
      nuvens:   'fundo-nuvens.png',
      nave:     'nave.png',
      ovni:     'ovni.png'
   };

   // Carregar todas
   for (var i in imagens) {
      var img = new Image();
      img.src = 'img/' + imagens[i];
      img.onload = carregando;
      totalImagens++;

      // Substituir o nome pela imagem
      imagens[i] = img;
   }
}
```

Temos então que criar a já conhecida função `carregando`, que vai monitorar o carregamento das imagens e iniciar a criação dos objetos quando todas estiverem prontas:

```
function carregando() {
   carregadas++;
   if (carregadas == totalImagens) iniciarObjetos();
}
```

A função `iniciarObjetos` iniciará os principais objetos do jogo, da forma como você já está acostumado a fazer:

```
function iniciarObjetos() {
   // Objetos principais
   animacao = new Animacao(context);
   teclado = new Teclado(document);
   colisor = new Colisor();
   espaco = new Fundo(context, imagens.espaco);
   estrelas = new Fundo(context, imagens.estrelas);
   nuvens = new Fundo(context, imagens.nuvens);
   nave = new Nave(context, teclado, imagens.nave);

   // Ligações entre objetos
   animacao.novoSprite(espaco);
   animacao.novoSprite(estrelas);
   animacao.novoSprite(nuvens);
   animacao.novoSprite(nave);

   colisor.novoSprite(nave);
   animacao.novoProcessamento(colisor);

   configuracoesIniciais();
}
```

Foi chamada a função `configuracoesIniciais`, que configura as velocidades dos fundos, posiciona a nave, configura o disparo pela tecla `Espaço` e inicia a animação:

```
function configuracoesIniciais() {
   // Fundos
   espaco.velocidade = 3;
   estrelas.velocidade = 5;
   nuvens.velocidade = 10;

   // Nave
   nave.x = canvas.width / 2 - imagens.nave.width / 2;
   nave.y = canvas.height - imagens.nave.height;
   nave.velocidade = 5;
```

```
    // Tiro
    teclado.disparou(ESPACO, function() {
        nave.atirar();
    });

    animacao.ligar();
}
```

Faça o teste: neste ponto você já deve ter a nave controlável e atirando, e o fundo rolando. Vamos agora configurar a criação dos inimigos como um processamento da animação. Ao final da função `configuracoesIniciais`, acrescente a chamada a `criacaoInimigos`:

```
function configuracoesIniciais() {
    // ...

    criacaoInimigos();
}
```

E vamos criar essa função. Ela cria um objeto sem construtor, com o método `processar`, e o insere como um processamento geral na animação:

```
function criacaoInimigos() {
    criadorInimigos = {
        processar: function() {

        }
    };

    animacao.novoProcessamento(criadorInimigos);
}
```

No método `processar` desse objeto, criaremos um inimigo a cada segundo. Mas, para isso, necessitamos saber o instante em que o último inimigo foi gerado. Este instante é guardado no atributo `ultimoOvni` e atualizado quando o tempo decorrido ultrapassar 1000 milissegundos:

```
function criacaoInimigos() {
    var criador = {
```

```
        ultimoOvni: new Date().getTime(),

        processar: function() {
            var agora = new Date().getTime();
            var decorrido = agora - this.ultimoOvni;

            if (decorrido > 1000) {
                novoOvni();
                this.ultimoOvni = agora;
            }
        }
    };

    animacao.novoProcessamento(criador);
}
```

Se passar um segundo desde a geração do último inimigo, é chamada a função `novoOvni`, que é muito semelhante à do capítulo anterior. Ela gera inimigos em posições e com velocidades aleatórias. Se não lembra como fazer isso, consulte o tópico 7.1.

```
function novoOvni() {
    var imgOvni = imagens.ovni;
    var ovni = new Ovni(context, imgOvni);

    // Mínimo: 5; máximo: 20
    ovni.velocidade =
        Math.floor( 5 + Math.random() * (20 - 5 + 1) );

    // Mínimo: 0;
    // máximo: largura do canvas - largura do ovni
    ovni.x =
        Math.floor(Math.random() *
                   (canvas.width - imgOvni.width + 1) );

    // Descontar a altura
    ovni.y = -imgOvni.height;

    animacao.novoSprite(ovni);
```

```
    colisor.novoSprite(ovni);
}
```

Agora estamos muito mais organizados, com cada etapa da inicialização do jogo em uma função específica. Estamos prontos para implementar as novidades.

8.2 Animação cronometrada

Para dar um aspecto mais profissional a nosso jogo, devemos cronometrar as animações. Você pode ter notado que, às vezes, a velocidade oscila. Isto é comum em browsers e jogos que rodam em sistemas operacionais multitarefa: a CPU pode estar ocupada enquanto o jogo espera o momento de processar o próximo ciclo.

Para resolver isso, precisamos primeiro saber o tempo decorrido entre um ciclo e outro. No construtor da classe `Animacao`, crie os atributos `ultimoCiclo` (para guardar o instante do ciclo anterior, lido do relógio) e `decorrido` (para guardar o tempo decorrido entre o ciclo anterior e o atual):

```
function Animacao(context) {
    // ...
    this.ultimoCiclo = 0;
    this.decorrido = 0;
}
```

Para fazer os cálculos, obtemos o instante atual do relógio do computador (dado por `Date.getTime()` em milissegundos) e calculamos a diferença entre esse instante e o instante do ciclo anterior. Faremos isso no método `proximoFrame`, logo antes de processar os sprites:

```
proximoFrame: function() {
    // Posso continuar?
    if ( ! this.ligado ) return;

    var agora = new Date().getTime();
    if (this.ultimoCiclo == 0) this.ultimoCiclo = agora;
    this.decorrido = agora - this.ultimoCiclo;
```

```
    // ...
}
```

No fim do método `proximoFrame`, atualize o atributo `ultimoCiclo`, imediatamente antes de chamar o próximo ciclo:

```
proximoFrame: function() {
    // ...

    // Atualizar o instante do último ciclo
    this.ultimoCiclo = agora;

    // Chamamos o próximo ciclo
    var animacao = this;
    requestAnimationFrame(function() {
        animacao.proximoFrame();
    });
},
```

Agora os sprites sabem quanto tempo levou entre um ciclo e outro! Vamos fazer o `Fundo` mover-se a uma velocidade constante. Para isso, modifique o método `atualizar`:

```
atualizar: function() {
    // Atualizar a posição de emenda
    this.posicaoEmenda +=
        this.velocidade * this.animacao.decorrido / 1000;

    // ...
},
```

Fórmula para animação cronometrada

O incremento da posição do sprite, em pixels, é dado pela fórmula:
`velocidade * tempoDecorrido / 1000`
Sendo:

- *velocidade* em pixels por segundo;
- *tempoDecorrido* em segundos (como o tempo dado por `Date.getTime()` é em milissegundos, dividimos esse valor por 1000).

Note que agora o fundo move-se em velocidade constante, porém bem devagar, pois passamos a trabalhar com pixels por segundo. Podemos ajustar novas velocidades com valores maiores na função `configuracoesIniciais` da página HTML. Você pode fazer vários testes e atribuir os valores que desejar, dependendo da sensação de velocidade que quer passar. Como as nuvens estão mais próximas, dei a elas a maior velocidade, mas nada impede que coloquemos as estrelas em primeiro plano, por exemplo:

```
function configuracoesIniciais() {
   // Fundos
   espaco.velocidade = 60;
   estrelas.velocidade = 150;
   nuvens.velocidade = 500;

   // ...
}
```

Vamos cronometrar também o movimento da `Nave`. Aplique a fórmula em seu método `atualizar`:

```
atualizar: function() {
   var incremento =
```

```
            this.velocidade * this.animacao.decorrido / 1000;

        // Use a variável incremento em todas as mudanças de x e y
        if (this.teclado.pressionada(SETA_ESQUERDA) && this.x > 0)
            this.x -= incremento;

        // ...
}
```

Na página HTML, dei a ela a velocidade de 200 pixels por segundo:

```
// Nave
nave.x = canvas.width / 2 - imagens.nave.width / 2;
nave.y = canvas.height - imagens.nave.height;
nave.velocidade = 200;
```

Exercício: faça você mesmo o ajuste para o `Ovni` e o `Tiro`! Dê-lhes as velocidades que desejar. A do `Tiro` é configurada no construtor, e as dos `Ovni`s, no método `novoOvni` da página. **Dica:** no pacote de download, a página já está com a solução implementada.

8.3 Animando a nave com spritesheets

No pacote de download do livro há o arquivo `nave-spritesheet.png` (figura 8.1). Iremos usá-lo para melhorar o aspecto de nossa nave. Convenhamos, ela está muito parada... nem o fogo em sua cauda se mexe!

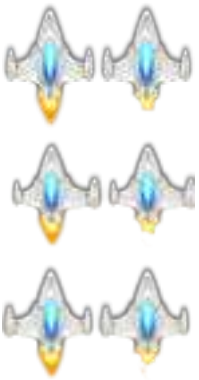


Figura 8.1: Spritesheet para a nave

No capítulo 4, definimos que as linhas da spritesheet representam diferentes estados do sprite. Aqui temos a nave parada, movendo-se para a esquerda e movendo-se para a direita. Em uma linha, a animação ocorre avançando as colunas. Em cada uma destas linhas, há duas colunas que animam o fogo na cauda.

Iniciando o teste

Para começar a programar com spritesheets, primeiro acrescente a referência ao arquivo `spritesheet.js`, criado no capítulo 4:

```
<script src="spritesheet.js"></script>
```

Na função `carregarImagens`, mude a imagem da nave para o arquivo `nave-spritesheet.png`:

```
imagens = {
   // ...
   nave:     'nave-spritesheet.png',
   // ...
};
```

Agora não podemos mais usar as dimensões da imagem para posicionar a nave. Em `configuracoesIniciais`, vamos passar valores absolutos referentes a cada quadro (considerando que cada nave tem 36x48 pixels):

```
function configuracoesIniciais() {
   // ...

   // Nave
   nave.x = canvas.width / 2 - 18;  // 36 / 2
   nave.y = canvas.height - 48;
   nave.velocidade = 5;

   // ...
}
```

No construtor da classe `Nave`, vamos iniciar o objeto que controla a spritesheet. Usaremos inicialmente a linha zero, que representa a nave parada. O intervalo entre um quadro e outro pode ser ajustado aos poucos conforme o seu gosto:

```
function Nave(context, teclado, imagem) {
   // ...
   this.spritesheet = new Spritesheet(context, imagem, 3, 2);
   this.spritesheet.linha = 0;
   this.spritesheet.intervalo = 100;
}
```

Altere agora o método `desenhar` para usar a spritesheet. Para definir qual a linha a ser animada, lemos o estado das setas do teclado:

```
desenhar: function() {
   if (this.teclado.pressionada(SETA_ESQUERDA))
      this.spritesheet.linha = 1;
   else if (this.teclado.pressionada(SETA_DIREITA))
      this.spritesheet.linha = 2;
   else
      this.spritesheet.linha = 0;

   this.spritesheet.desenhar(this.x, this.y);
   this.spritesheet.proximoQuadro();
},
```

Também precisamos alterar o método `atualizar`, pois ele também está usando as dimensões da imagem para não deixá-la passar da borda

do Canvas. Nós substituímos as referências a `this.imagem.width` e `this.imagem.height` pelos valores absolutos 36 e 48, respectivamente:

```
atualizar: function() {
   // ...

   if (this.teclado.pressionada(SETA_DIREITA) &&
            this.x < this.context.canvas.width - 36)
      this.x += this.velocidade;

   // ...

   if (this.teclado.pressionada(SETA_ABAIXO) &&
            this.y < this.context.canvas.height - 48)
      this.y += this.velocidade;
},
```

Por último, perceba que o tiro sai um pouco deslocado, pois sua posição é calculada pelo tamanho da imagem antiga. Vamos ajustar para um valor absoluto, no construtor da classe `Tiro`. Também aproveitei o momento para mudar a cor e deixá-lo um pouco menor:

```
function Tiro(context, nave) {
   this.context = context;
   this.nave = nave;

   // Posicionar o tiro no bico da nave
   this.largura = 3;
   this.altura = 10;
   this.x = nave.x + 18;  // 36 / 2
   this.y = nave.y - this.altura;
   this.velocidade = 10;

   this.cor = 'yellow';
}
```

Experimente jogar e perceba que o movimento da nave ganhou mais dinamismo!

8.4 Criando explosões

Vamos aproveitar o embalo das spritesheets e criar explosões. Usaremos o arquivo `explosao.png`, que contém a spritesheet na figura 8.2:

Figura 8.2: Spritesheet para explosão

Também será preciso acrescentar uma referência ao script `explosao.js`, a ser criado logo mais:

```
<script src="explosao.js"></script>
```

Na função `carregarImagens`, acrescente a imagem da explosão:

```
imagens = {
   // ...
   ovni:     'ovni.png',  // uma vírgula aqui
   explosao: 'explosao.png'
};
```

No método `colidiuCom` da classe `Ovni`, onde este é destruído pelo `Tiro`, vamos criar uma explosão:

```
colidiuCom: function(outro) {
   // Se colidiu com um Tiro, os dois desaparecem
   if (outro instanceof Tiro) {
      // ...

      var explosao = new Explosao(this.context,
         this.imgExplosao, this.x, this.y);
      this.animacao.novoSprite(explosao);
   }
}
```

Os objetos que explodem precisam receber a imagem da explosão, para poder criar os sprites. Mude o construtor do `Ovni`:

```
function Ovni(context, imagem, imgExplosao) {
   // ...
   this.imgExplosao = imgExplosao;
}
```

E passe a imagem na função `novoOvni` da página HTML:

```
var ovni = new Ovni(context, imgOvni, imagens.explosao);
```

Agora, crie a classe `Explosao` no arquivo `explosao.js`:

```
function Explosao(context, imagem, x, y) {
   this.context = context;
   this.imagem = imagem;
   this.spritesheet = new Spritesheet(context, imagem, 1, 5);
   this.spritesheet.intervalo = 75;
   this.x = x;
   this.y = y;
}
Explosao.prototype = {
   atualizar: function() {

   },
   desenhar: function() {

   }
}
```

No método `desenhar`, nós desenhamos o quadro atual e animamos a spritesheet:

```
desenhar: function() {
   this.spritesheet.desenhar(this.x, this.y);
   this.spritesheet.proximoQuadro();
}
```

Mas assim a explosão ficará piscando na tela eternamente! O que queremos é que, quando todos os quadros forem exibidos, a explosão termine.

Mas como saber se a spritesheet chegou ao último quadro? Vamos criar nela uma funcionalidade com a qual ela própria avisa isto. No construtor da `Explosao`, crie um callback que recebe esse aviso:

```
function Explosao(context, imagem, x, y) {
   // ...
   var explosao = this;
   this.spritesheet.fimDoCiclo = function() {
      explosao.animacao.excluirSprite(explosao);
   }
}
```

Agora crie o atributo `fimDoCiclo` no construtor da `Spritesheet`:

```
function Spritesheet(context, imagem, linhas, colunas) {
   // ...
   this.fimDoCiclo = null;
}
```

E modifique o método `proximoQuadro()` para chamar esse callback quando voltar ao quadro zero:

```
proximoQuadro: function() {
   // ...

   if (this.coluna < this.numColunas - 1) {
      this.coluna++;
   }
   else {
      this.coluna = 0;

      // Avisar que acabou um ciclo!
      if (this.fimDoCiclo) this.fimDoCiclo();
   }

   // ...
},
```

Experimente jogar! O `Ovni` já explode ao ser atingido pelo tiro. Agora vamos explodir tanto a `Nave` quanto o `Ovni` quando eles colidirem. No método `colidiuCom` da `Nave`, vamos criar duas explosões:

```
colidiuCom: function(outro) {
   // Se colidiu com um Ovni...
   if (outro instanceof Ovni) {
      this.animacao.excluirSprite(this);
      this.animacao.excluirSprite(outro);
      this.colisor.excluirSprite(this);
      this.colisor.excluirSprite(outro);

      var exp1 = new Explosao(this.context, this.imgExplosao,
                              this.x, this.y);
      var exp2 = new Explosao(this.context, this.imgExplosao,
                              outro.x, outro.y);

      this.animacao.novoSprite(exp1);
      this.animacao.novoSprite(exp2);

      // Por enquanto tire o término da animação
   }
}
```

Claro, a `Nave` também precisará receber a imagem da explosão no construtor:

```
function Nave(context, teclado, imagem, imgExplosao) {
   // ...
   this.imgExplosao = imgExplosao;
}
```

Portanto, passe-a ao criar a nave (método `iniciarObjetos` da página HTML):

```
nave = new Nave(context, teclado, imagens.nave,
                imagens.explosao);
```

A `Nave` e o `Ovni` já explodem juntos. Gostaríamos de parar a animação e dar a mensagem "GAME OVER" somente quando essas explosões finalizarem. Portanto, o sprite `Explosao` também poderia receber um callback. Complete o `colidiuCom` da `Nave` para passar esse callback:

```
colidiuCom: function(outro) {
   // Se colidiu com um Ovni...
   if (outro instanceof Ovni) {
      // ...

      exp1.fimDaExplosao = function() {
         animacao.desligar();
         alert('GAME OVER');
      }
   }
}
```

E modifique o construtor da `Explosao` para chamar esse callback ao fim do ciclo da spritesheet:

```
function Explosao(context, imagem, x, y) {
   // ...

   var explosao = this;
   this.fimDaExplosao = null;
   this.spritesheet.fimDoCiclo = function() {
      explosao.animacao.excluirSprite(explosao);
      if (explosao.fimDaExplosao) explosao.fimDaExplosao();
   }
}
```

Não é legal? A `Spritesheet` informa à `Explosao` quando completou a animação dos quadros, e a `Explosao` se retira da animação e notifica a quem interessar.

Brinque um pouco, acho que você merece!

Figura 8.3: Agora as colisões têm emoção!

8.5 Pausando o jogo

Figura 8.4: Jogo pausado com indicativo na tela

Nós já temos pausa implementada em nosso game engine: os métodos `ligar` e `desligar` da classe `Animacao`. No entanto, é preciso tratar alguns acontecimentos que estão ocorrendo na animação, ao se pausar o jogo.

Vamos definir que o jogador pausará o jogo através da tecla `Enter`, cujo código é 13 (se preferir usar outra tecla, fique à vontade).

No início do arquivo `teclado.js`, acrescente uma variável para guardar o código do `Enter`:

```
// Códigos de teclas - aqui vão todos os que forem necessários
var SETA_ESQUERDA = 37;
var SETA_ACIMA = 38;
var SETA_DIREITA = 39;
var SETA_ABAIXO = 40;
```

```
var ESPACO = 32;
var ENTER = 13;
```

Agora, no método `configuracoesIniciais`, configure a ação da tecla `Enter`:

```
// Pausa
teclado.disparou(ENTER, pausarJogo);
```

E crie a função `pausarJogo`, que é quem fará o trabalho:

```
function pausarJogo() {
   if (animacao.ligado)
      animacao.desligar();
   else
      animacao.ligar();
}
```

Faça o teste: neste ponto o jogo já deve pausar e despausar. No entanto, você poderá notar alguns bugs. Primeiro, pause enquanto um disco voador está na tela, e em seguida despause: ele some! Isso ocorre porque sua animação está cronometrada, e o tempo da pausa está sendo contado para calcular seu movimento. Quando a animação reinicia, o disco já deve estar longe, e assim o jogo o posiciona.

Para resolver isso, na classe `Animacao`, no método `ligar`, reinicie o atributo `ultimoCiclo` para zero. Isto fará a contagem de tempo ser reiniciada:

```
ligar: function() {
   this.ultimoCiclo = 0;
   this.ligado = true;
   this.proximoFrame();
},
```

Um outro problema é: funções de disparo (no caso, o tiro) continuam sendo possíveis! Isto não ocorre com a movimentação pelas setas, pois são lidas no método `atualizar` da `Nave`, dentro do loop de animação (que estaria pausado).

Modifiquemos a função para desativar ou reativar a tecla `Espaço` (tiro) conforme pausamos e despausamos o jogo:

```
function pausarJogo() {
   if (animacao.ligado) {
      animacao.desligar();
      ativarTiro(false);
   }
   else {
      animacao.ligar();
      ativarTiro(true);
   }
}
```

A função `ativarTiro` vai modificar os eventos do teclado, configurando a tecla `Espaço`:

```
function ativarTiro(ativar) {
   if (ativar) {
      teclado.disparou(ESPACO, function() {
         nave.atirar();
      });
   }
   else
      teclado.disparou(ESPACO, null);
}
```

Para não ficarmos com código repetido, em `configuracoesIniciais`, remova as linhas:

```
// Remova estas linhas
// Tiro
teclado.disparou(ESPACO, function() {
   nave.atirar();
});
```

E coloque esta no lugar:

```
// Tiro
ativarTiro(true);
```

Também ocorre que um inimigo é gerado imediatamente após a despausa, pois estamos gerando um inimigo por segundo e a pausa faz esse tempo se

exceder. Antes de ligar a animação, reinicie o momento da geração do último inimigo para o instante atual, para evitar que isso ocorra:

```
function pausarJogo() {
   // ...

   else {
      criadorInimigos.ultimoOvni = new Date().getTime();
      animacao.ligar();
      ativarTiro(true);
   }
}
```

Quantos detalhes uma simples pausa nos obriga a tratar! Para ficar mais interessante, vamos exibir o texto "Pausado" na tela:

```
function pausarJogo() {
   if (animacao.ligado) {
      animacao.desligar();
      ativarTiro(false);

      // Escrever "Pausado"
      context.save();
      context.fillStyle = 'white';
      context.strokeStyle = 'black';
      context.font = '50px sans-serif';
      context.fillText("Pausado", 160, 200);
      context.strokeText("Pausado", 160, 200);
      context.restore();
   }
   else {
      criadorInimigos.ultimoOvni = new Date().getTime();
      animacao.ligar();
      ativarTiro(true);
   }
}
```

Use sua imaginação! Coloque qualquer imagem, texto, ou combinação dos dois. Pode até instanciar outra `Animacao`, colocar alguns sprites específicos e exibir alguns movimentos durante a pausa.

8.6 Sons e música de fundo

Esta parte é muito fácil de implementar. O HTML5, além do Canvas, possui uma API de áudio muito simples. No pacote de arquivos, na pasta deste capítulo (`08`), há uma subpasta com o nome `snd` contendo alguns arquivos de som, em formato MP3.

> **Bancos de sons grátis e direitos autorais**
>
> Jamais use sons de terceiros sem a devida autorização! Criadores de jogos independentes, sem meios ou conhecimentos para gerar seus próprios sons, podem contar com inúmeros bancos de sons gratuitos espalhados pela internet.
>
> Os sons para o jogo deste livro foram obtidos em http://freesound.org.

Primeiro, vamos produzir o som dos tiros. Abra o arquivo `tiro.js` e coloque as instruções para carregar o som logo no início do arquivo. Instanciamos um objeto `Audio` (presente na API do HTML5), setamos seu atributo `src` para o arquivo `tiro.mp3` (presente no pacote de download) e ajustamos o volume como um valor entre 0 e 1. Por último, chamamos o método `load` para iniciar o carregamento do arquivo, evitando que seja carregado somente no momento da reprodução:

```
var SOM_TIRO = new Audio();
SOM_TIRO.src = 'snd/tiro.mp3';
SOM_TIRO.volume = 0.2;
SOM_TIRO.load();
```

Depois, vamos reproduzir esse som cada vez que um tiro é criado. No construtor, rebobinamos o som ajustando o atributo `currentTime` para zero. Esse atributo indica o instante atual de reprodução, dado em segundos. Em seguida, basta chamar o método `play`:

```
function Tiro(context, nave) {
   // ...
```

```
    SOM_TIRO.currentTime = 0.0;
    SOM_TIRO.play();
}
```

Os tiros já fazem barulho! Vamos fazer o mesmo procedimento para as explosões. No início do arquivo `explosao.js`, coloque as instruções:

```
var SOM_EXPLOSAO = new Audio();
SOM_EXPLOSAO.src = 'snd/explosao.mp3';
SOM_EXPLOSAO.volume = 0.4;
SOM_EXPLOSAO.load();
```

E no construtor, mande rebobinar e reproduzir, da mesma forma que com o `Tiro`:

```
function Explosao(context, imagem, x, y) {
    // ...

    SOM_EXPLOSAO.currentTime = 0.0;
    SOM_EXPLOSAO.play();
}
```

Vamos também colocar uma música de fundo para dar mais emoção ao jogo. No início da página HTML, acrescente a variável `musicaAcao`:

```
// Variáveis principais
var imagens, animacao, teclado, colisor, nave, criadorInimigos;
var totalImagens = 0, carregadas = 0;
var musicaAcao;
```

Em seguida, logo após a chamada para `carregarImagens`, coloque também uma chamada para `carregarMusicas`:

```
// Começa carregando as imagens e músicas
carregarImagens();
carregarMusicas();
```

E crie a função `carregarMusicas`. Aqui, setamos o atributo `loop` para `true`, fazendo com que a música repita incessantemente durante a ação do jogo.

```
function carregarMusicas() {
    musicaAcao = new Audio();
    musicaAcao.src = 'snd/musica-acao.mp3';
    musicaAcao.load();
    musicaAcao.volume = 0.8;
    musicaAcao.loop = true;
    musicaAcao.play();
}
```

Viu como é bem fácil? Divirta-se mais um pouco!

Dica: você pode parar a música no momento da pausa usando o método `pause`.

8.7 Tela de loading

Figura 8.5: Indique para o jogador que o jogo está carregando

Vamos modificar a função `carregando` da página HTML para incrementar uma barra conforme as imagens vão sendo carregadas. Primeiro, como vamos desenhar, guarde a configuração atual do contexto (método `save`) e crie um fundo. Aqui, fiz um fundo simples, com a imagem do espaço. Se tiver vontade faça um desenho mais elaborado. Sinta-se livre!

```
function carregando() {
    context.save();

    // Fundo
    context.drawImage(imagens.espaco, 0, 0, canvas.width,
                      canvas.height);

    // continua ...
}
```

Em seguida, vamos criar o texto “Carregando":

```
function carregando() {
    // ...

    // Texto "Carregando"
    context.fillStyle = 'white';
    context.strokeStyle = 'black';
    context.font = '50px sans-serif';
    context.fillText("Carregando...", 100, 200);
    context.strokeText("Carregando...", 100, 200);

    // continua ...
}
```

O próximo passo é desenhar a barra. Neste ponto, precisamos incrementar a variável `carregadas`, para poder calcular a largura atual da barra:

```
function carregando() {
    // ...

    // Barra de loading
    carregadas++;
```

```
    var tamanhoTotal = 300;
    var tamanho = carregadas / totalImagens * tamanhoTotal;
    context.fillStyle = 'yellow';
    context.fillRect(100, 250, tamanho, 50);

    // continua ...
}
```

Por último, restauramos as configurações anteriores do contexto (método `restore`) e, em vez de iniciar o jogo imediatamente, vamos mostrar um link "Jogar", a ser criado logo adiante, para o jogador clicar quando estiver pronto:

```
function carregando() {
    // ...

    context.restore();

    if (carregadas == totalImagens) {
        iniciarObjetos();
        mostrarLinkJogar();
    }
}
```

Como o jogo não vai iniciar automaticamente, tire o comando que inicia a animação da função `configuracoesIniciais`:

```
// Remova esta linha
animacao.ligar();
```

E também a chamada ao método `play` da função `carregarMusicas`:

```
// Remova esta linha
musicaAcao.play();
```

Crie então um link para iniciar o jogo. Logo após a tag `<canvas>`, insira esse link, com uma chamada para a função `iniciarJogo`:

```
<a id="link_jogar" href="javascript: iniciarJogo()">Jogar</a>
```

Vamos configurar sua aparência e posição via CSS. Você pode configurar esse botão da forma que quiser, e não é o foco deste livro ficar detalhando formatações em CSS. Caso não tenha muita prática, aí está uma formatação sugerida. No pacote de downloads, existe a imagem `botao-jogar.png`, que usei como fundo.

Na seção `<head>` do documento (pode ser após os scripts), crie uma tag `<style>`. O mais importante é que o botão inicie oculto e tenha posicionamento absoluto, para podermos colocá-lo por cima do Canvas:

```
<style>
#link_jogar {
   /* Inicia oculto */
   display: none;

   /* Cores e fundo */
   color: yellow;
   background: url(img/botao-jogar.png);

   /* Fonte */
   font-size: 20px;
   font-family: sans-serif;

   /* Sem sublinhado e com sombra */
   text-decoration: none;
   text-shadow: 2px 2px 5px black;

   /* Posicionamento */
   position: absolute;
   left: 220px;
   top: 330px;

   /* A imagem do botão é 72x72, descontamos os paddings */
   width: 52px;
   height: 26px;
   padding: 23px 10px;
}
</style>
```

E crie a função `mostrarLinkJogar`, que mandará mostrar esse link, modificando seu atributo CSS `display` para o valor `block`.

```
function mostrarLinkJogar() {
   document.getElementById('link_jogar').style.display =
      'block';
}
```

A função `iniciarJogo`, chamada pelo link, irá esconder o link "Jogar", iniciar a música e ligar a animação:

```
function iniciarJogo() {
   document.getElementById('link_jogar').style.display =
      'none';
   musicaAcao.play();
   animacao.ligar();
}
```

Faça o teste! Enquanto você estiver testando o jogo em seu computador, a tela carregará bem rapidamente. No entanto, ela fará (muita) diferença quando hospedarmos nosso jogo na web.

Ainda temos um problema: a tecla `Enter` dá início ao jogo sem fazer sumir o botão Jogar! Isso ocorre porque ela está despausando o jogo em um momento em que a animação não está rodando.

Para corrigir isso, vamos tirar as seguintes linhas de `configuracoesIniciais`:

```
// Remova estas linhas
// Tiro
ativarTiro(true);

// E estas também
// Pausa
teclado.disparou(ENTER, pausarJogo);
```

Vamos ativá-las em `iniciarJogo`:

```
function iniciarJogo() {
   // Tiro
```

```
    ativarTiro(true);

    // Pausa
    teclado.disparou(ENTER, pausarJogo);

    // ...
}
```

8.8 Vidas extras

Figura 8.6: Dê mais chances ao jogador com vidas extras!

Programar vidas extras não é tão complicado, tudo é questão de mexer nos algoritmos da nave. Para facilitar, vamos criar um atributo de callback onde a `Nave` nos informa quando acabaram as vidas. No construtor, crie esse callback e o atributo `vidasExtras`:

```
function Nave(context, teclado, imagem, imgExplosao) {
    // ...
    this.acabaramVidas = null;
    this.vidasExtras = 3;
}
```

No método `colidiuCom`, onde detectamos sua colisão mortal, retire as linhas que mandam finalizar o jogo:

```
// Retire estas linhas:
exp1.fimDaExplosao = function() {
    animacao.desligar();
    alert('GAME OVER');
}
```

E substitua-as pelo código a seguir. Nós vamos decrementar o atributo `vidasExtras`, e verificar se as vidas acabaram (e notificar, se for o caso).

Caso o jogo continue, nós recolocamos a nave na animação e no colisor e a reposicionamos:

```
var nave = this;
exp1.fimDaExplosao = function() {
   nave.vidasExtras--;

   if (nave.vidasExtras < 0) {
      if (nave.acabaramVidas) nave.acabaramVidas();
   }
   else {
      // Recolocar a nave no engine
      nave.colisor.novoSprite(nave);
      nave.animacao.novoSprite(nave);

      nave.posicionar();
   }
}
```

O posicionamento da nave é exatamente o que já estamos fazendo na função `configuracoesIniciais` da página HTML. Retire as linhas:

```
// Nave
// Retire as duas linhas abaixo
nave.x = canvas.width / 2 - 18;  // 36 / 2
nave.y = canvas.height - 48;
```

E coloque no lugar uma chamada ao método `posicionar`:

```
// Nave
nave.posicionar();
nave.velocidade = 200;
```

Vamos criar o método na `Nave`, fazendo essa mesma tarefa:

```
posicionar: function() {
   var canvas = this.context.canvas;
   this.x = canvas.width / 2 - 18;  // 36 / 2
   this.y = canvas.height - 48;
}
```

Por fim, em `configuracoesIniciais`, insira as linhas que finalizam o jogo, respondendo ao callback `acabaramVidas` da `Nave`:

```
function configuracoesIniciais() {
   // ...

   // Game Over
   nave.acabaramVidas = function() {
      animacao.desligar();
      alert('GAME OVER');
   }
}
```

Exercício: implemente, através da leitura do teclado, um *cheat code* que dê mais vidas ao jogador!

Um mostrador de vidas

Legal, nossa nave já tem vidas extras! Vamos criar agora o arquivo `painel.js`, para mostrar essas vidas extras. Seu esqueleto fica:

```
// arquivo: painel.js
function Painel(context, nave) {
   this.context = context;
   this.nave = nave;
}
Painel.prototype = {
   atualizar: function() {

   },
   desenhar: function() {

   }
}
```

Claro, insira o script na seção `<head>`:

```
<script src="painel.js"></script>
```

Na função `iniciarObjetos`, crie o painel e insira-o na animação. Optei por deixar a nave por último, para que ela sempre fique acima do painel:

```
function iniciarObjetos() {
   // Objetos principais
   // ...
   painel = new Painel(context, nave);

   // Ligações entre objetos
   // ...
   animacao.novoSprite(painel);
   animacao.novoSprite(nave);

   // ...
}
```

No construtor do `Painel`, vamos criar uma spritesheet a partir da imagem da nave. Nós usaremos somente a primeira imagem, que corresponde à nave parada:

```
function Painel(context, nave) {
   // ...
   this.spritesheet =
      new Spritesheet(context, nave.imagem, 3, 2);
   this.spritesheet.linha = 0;
   this.spritesheet.coluna = 0;
}
```

No método `desenhar`, obtemos o número de vidas extras e desenhamos a imagem esse número de vezes:

```
desenhar: function() {
   var x = 20;
   var y = 20;

   for (var i = 1; i <= this.nave.vidasExtras; i++) {
      this.spritesheet.desenhar(x, y);
      x += 40;
   }
}
```

Eu achei que os desenhos das vidas estão muito grandes. Vou usar aqui o método `scale` do *context* para desenhá-los com metade do tamanho:

```
desenhar: function() {
   // Reduz o desenho pela metade
   this.context.scale(0.5, 0.5);

   var x = 20;
   var y = 20;

   for (var i = 1; i <= this.nave.vidasExtras; i++) {
      this.spritesheet.desenhar(x, y);
      x += 40;
   }

   // Torna a dobrar
   this.context.scale(2, 2);
}
```

Você pode usar o tamanho que preferir, ou mesmo usar um arquivo de imagem apenas para isso, em tamanho menor. Mas com o `scale` é bem mais econômico, não acha?

8.9 Pontuação (*score*)

Aproveitando que temos um painel mostrador de vidas, podemos criar nele um atributo `pontuacao`. Em sua função construtora, coloque a instrução:

```
this.pontuacao = 0;
```

Em `configuracoesIniciais`, vamos criar um tratador geral de colisões que verifica se é um `Tiro` e um `Ovni` que estão colidindo. Em caso positivo, incrementamos essa pontuação no painel:

```
function configuracoesIniciais() {
   // ...

   // Pontuação
   colisor.aoColidir = function(o1, o2) {
      // Tiro com Ovni
      if ( (o1 instanceof Tiro && o2 instanceof Ovni) ||
           (o1 instanceof Ovni && o2 instanceof Tiro) )
```

```
            painel.pontuacao += 10;
            // Use o incremento que desejar
        }
    }
```

Agora, modifique o método `desenhar` do painel para escrever a pontuação ao lado dos ícones das vidas:

```
desenhar: function() {
    // ...

    // Para facilitar um pouco...
    var ctx = this.context;

    // Pontuação
    ctx.save();
    ctx.fillStyle = 'white';
    ctx.font = '18px sans-serif';
    ctx.fillText(this.pontuacao, 100, 27);
    ctx.restore();
}
```

Por último, não se esqueça de zerar o painel cada vez que o jogo inicia, do contrário a pontuação fica lá para a próxima partida. Em `iniciarJogo`, coloque o comando:

```
painel.pontuacao = 0;
```

8.10 Tela de Game Over

Figura 8.7: Que pena... fim de jogo!

Vamos finalizar nosso jogo criando uma tela de Game Over! Em `configuracoesIniciais`, tire aquele alerta horrível e troque por uma chamada à função `gameOver`:

```
function configuracoesIniciais() {
   // ...

   // Game Over
   nave.acabaramVidas = function() {
      animacao.desligar();
      gameOver();
   }
}
```

Nessa função, vamos desativar as teclas de disparo, parar a música e criar uma tela mais adequada. O link "Jogar" deve voltar a aparecer, e a nave deve

ser recolocada no game engine (pois foi tirada no momento da colisão):

```
function gameOver() {
    // Tiro
    ativarTiro(false);

    // Pausa
    teclado.disparou(ENTER, null);

    // Parar a música e rebobinar
    musicaAcao.pause();
    musicaAcao.currentTime = 0.0;

    // Fundo
    context.drawImage(imagens.espaco, 0, 0, canvas.width,
                      canvas.height);

    // Texto "Game Over"
    context.save();
    context.fillStyle = 'white';
    context.strokeStyle = 'black';
    context.font = '70px sans-serif';
    context.fillText("GAME OVER", 40, 200);
    context.strokeText("GAME OVER", 40, 200);
    context.restore();

    // Volta o link "Jogar"
    mostrarLinkJogar();

    // Restaurar as condições da nave
    nave.vidasExtras = 3;
    nave.posicionar();
    animacao.novoSprite(nave);
    colisor.novoSprite(nave);
}
```

Você perceberá uma coisa: `Ovni`s que sobraram na animação, continuam lá para o próximo jogo! Ao fim de `gameOver`, chame uma nova função, `removerInimigos`:

```
function gameOver() {
   // ...

   removerInimigos();
}
```

Nesta função, nós mandamos excluir todos os `Ovni`s da animação:

```
function removerInimigos() {
   for (var i in animacao.sprites) {
      if (animacao.sprites[i] instanceof Ovni)
         animacao.excluirSprite(animacao.sprites[i]);
   }
}
```

Finalmente, reinicie o instante da geração de inimigos em `iniciarJogo`, para que não surja um `Ovni` de repente quando o jogo for reiniciado. Faça isto antes de ligar a animação:

```
function iniciarJogo() {
   criadorInimigos.ultimoOvni = new Date().getTime();

   // ...
}
```

Pode soltar fogos de artifício! Temos um jogo simples, embora completo em termos de animação, engine e algoritmos.

Exercícios

Agora ficará por sua conta melhorar o jogo! Aqui estão algumas sugestões do que você pode fazer:

- Criar novos tipos de inimigos;
- Fases e cenários diferentes;
- Tipos diferentes de naves e tiros;
- Coleta de itens;
- E o que mais sua imaginação permitir!

Com os conhecimentos que você adquiriu, essas tarefas não deverão ser tão árduas assim. Se você sentir que não tem bom domínio do game engine, sugiro que refaça a criação do jogo, quantas vezes forem necessárias, até sentir-se seguro.

Capítulo 9

Publique seu jogo e torne-o conhecido

9.1 Hospede-o em um serviço gratuito

Talvez você, sendo desenvolvedor web, já tenha os meios de ter seu jogo publicado na web. No entanto, para tirar quaisquer eventuais dúvidas, ou caso você seja iniciante nesta matéria, quero mostrar um meio de você colocar seu jogo na internet, usando um dos inúmeros serviços de hospedagem gratuita existentes.

Nosso jogo não é mais do que uma página web estática, usando apenas tecnologias do lado cliente (um pouco de HTML e CSS, e muito JavaScript). Desta forma, pode ser colocado na web como qualquer site comum.

Escolhi o *Hostinger* (http://www.hostinger.com.br) , por considerá-lo fá-

cil de usar e já estar habituado a ele. No entanto, você pode escolher outro serviço, se preferir. Acesse o site e clique no link `Criar conta`:

Hospedagens gratuitas

Alguns exemplos de hospedagens gratuitas disponíveis:

- Hostinger: http://www.hostinger.com.br
- Hospedagratis.net: http://hospedagratis.net
- Qlix: http://www.qlix.com.br

Figura 9.1: Criando uma conta no Hostinger

O formulário de cadastro não é muito diferente dos formulários mais básicos existentes na web:

Figura 9.2: Cadastro no Hostinger

Após isto, é preciso acessar seu e-mail e clicar no link de ativação enviado pela Hostinger. Caso não apareça, confira sua pasta de spam ou lixo eletrônico.

Do link, você irá direto para a escolha do plano de hospedagem. Para pequenos aplicativos, podemos usar o plano gratuito sem medo:

Figura 9.3: Você está 'comprando' o plano gratuito pelo incrível preço de zero reais!

Em seguida, você tem a opção de usar um subdomínio gratuito, ao invés

de adquirir um nome de domínio (lembre-se de que, para ter seu endereço próprio na internet, é preciso pagar a taxa de registro). Foi esta opção que usei, embora eu imagine que você, como futuro desenvolvedor de jogos, vá um dia querer ter o domínio *www.jogosdo(seunomeaqui).com.br*:

Figura 9.4: Criando o subdomínio

A próxima tela será apenas uma confirmação dos dados do domínio. Digite as letras e aceite os termos como de costume.

O domínio está criado! Clique em seu nome e em seguida aparecerá o ícone `Gerenciar`:

Figura 9.5: Abra o domínio e entre em Gerenciar

Você terá que rolar um pouco a página para encontrar o `Gerenciador de Arquivos`:

Figura 9.6: Ícone do Gerenciador de Arquivos

Se o serviço solicitar uma confirmação para instalar o gerenciador, clique em `Instalar`.

Crie as subpastas do jogo. Use o botão `Nova Pasta` e crie as pastas `img` e `snd`:

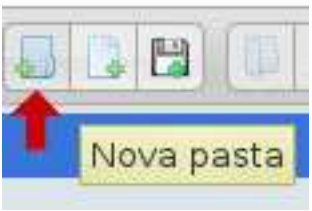


Figura 9.7: Criando uma nova pasta na hospedagem

Figura 9.8: Crie as pastas img e snd

Use o botão `Fazer Upload` e suba os arquivos da pasta principal. Em seguida, entre nas pastas `img` e `snd` e repita o processo. Você pode subir vários arquivos de uma vez.

Figura 9.9: Botão de upload de arquivos

Renomeie o arquivo `jogo-definitivo.html` para `index.html`. É por esse nome que o servidor reconhece a página inicial de seu site:

Figura 9.10: A página principal deve se chamar index.html

Por último, delete a antiga página principal colocada pela Hostinger (arquivo `default.php`).

Pronto, pode navegar no site recém-criado e publicar seu URL onde quiser!

Figura 9.11: Jogo publicado em hospedagem gratuita

9.2 Linkando com as redes sociais

Obtendo uma ID do Facebook

Para poder interagir com o Facebook em suas aplicações, você deve possuir um **App ID**, que é um código de cadastro no Facebook associado com seu aplicativo.

Primeiro, faça o login no Facebook como de costume. Em seguida, acesse http://developers.facebook.com e abra o menu `Aplicativos`. Se você ainda não é desenvolvedor, será apresentada a opção `Register as a Developer`:

Figura 9.12: Registre-se como desenvolvedor no Facebook

Sua conta será confirmada como desenvolvedor. Agora você tem a opção `Create a New App`:

Figura 9.13: Criando um aplicativo no Facebook

Em seguida, você terá que preencher os dados como o nome, uma identificação (*namespace*) que deve ser única para cada aplicativo, a categoria etc.

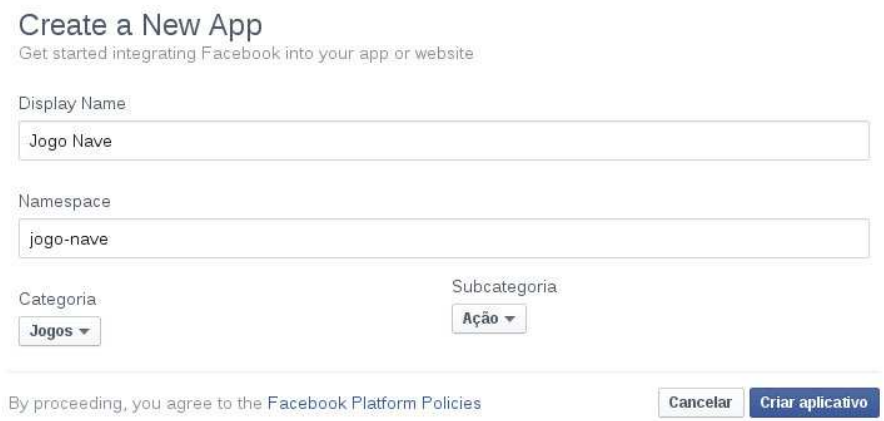


Figura 9.14: Cadastre seu aplicativo

Pronto, você já tem uma ID! Guarde o código em algum lugar seguro.

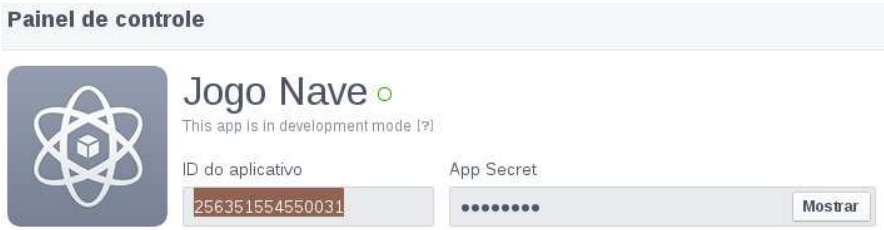


Figura 9.15: ID de aplicação no Facebook

Agora, temos que configurar a URL de nosso site para evitar problemas de permissão. Vá ao link `Configurações` e clique no botão `Adicionar Plataforma`. Selecione `Site` e preencha a URL de seu site no campo que foi aberto:

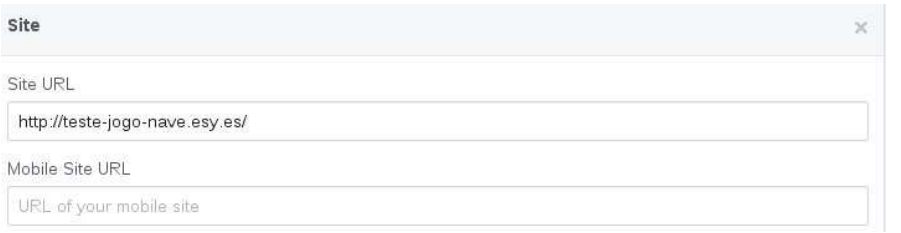


Figura 9.16: Configure a URL de seu site

Por último, no campo `App Domains`, insira o endereço que você acabou de configurar (o Facebook já lista automaticamente as URLs configuradas):

Figura 9.17: Configure a URL no campo App Domains

Estamos prontos para inserir funcionalidades do Facebook em nosso jogo!

Inicializando a API

O código de inicialização da API pode ser obtido do próprio Facebook, no endereço http://developers.facebook.com/docs/javascript/quickstart/v2.0. A própria documentação recomenda que seja colado logo após a abertura da tag `<body>`:

```
<body>

    <script>
      window.fbAsyncInit = function() {
        FB.init({
          // Esta é sua ID
          appId      : '256351554550031',
```

```
          xfbml      : true,
          version    : 'v2.0'
        });
      };

      (function(d, s, id){
         var js, fjs = d.getElementsByTagName(s)[0];
         if (d.getElementById(id)) {return;}
         js = d.createElement(s); js.id = id;
         js.src = "http://connect.facebook.net/pt_BR/sdk.js";
         fjs.parentNode.insertBefore(js, fjs);
       }(document, 'script', 'facebook-jssdk'));
    </script>

    ...

</body>
```

Botão Compartilhar

Em http://developers.facebook.com/docs/plugins/share-button, você pode gerar um botão "Compartilhar" para sua página. O código gerado é semelhante ao mostrado a seguir. Coloque-o no local que achar melhor, dando-lhe bastante destaque na página:

```
<div id="botao_compartilhar"
     class="fb-share-button"
     data-href="http://sua_hospedagem_aqui"
     data-type="button_count"></div>
```

Compartilhando a pontuação

Uma coisa muito interessante é podermos postar qualquer informação programaticamente. Por exemplo, podemos colocar um link para postar o score na tela de Game Over!

Primeiro, criamos esse link:

```
<a id="postar_pontuacao" href="javascript: postarPontuacao()">
  Compartilhar Pontuação
```

```
</a>
```

Este link deve começar escondido, portanto, crie uma formatação CSS que o oculte:

```
<style>

...

#postar_pontuacao {
   /* Ocultar o link */
   display: none;

   /* Visual a gosto! */
   color: yellow;
   position: absolute;
   left: 180px;
   top: 410px;
}
</style>
```

E mande-o aparecer na função `gameOver`:

```
function gameOver() {
   // ...

   // Link de postar pontuação
   document.getElementById('postar_pontuacao')
      .style.display = 'block';
}
```

A função `postarPontuacao` chama a API do Facebook para abrir uma janela que compartilha texto:

```
function postarPontuacao() {
    // Aqui obteríamos a pontuação do jogo
    var pontuacao = 100000;

    // API de feed do Facebook
    FB.ui(
```

```
      {
       method: 'feed',
       name: 'Jogo de Nave',
       caption: 'Jogue e compartilhe com seus amigos!',
       description: (
          'Minha pontuação foi: ' + painel.pontuacao
       ),
       link: 'http://sua_hospedagem_aqui'
      },
      function(response) {}
    );
}
```

Por último, mande o link desaparecer na função `iniciarJogo`:

```
function iniciarJogo() {
   document.getElementById('postar_pontuacao').style.display =
      'none';

   // ...
}
```

Dê ao link um visual bem agradável e veja-o na tela de Game Over:

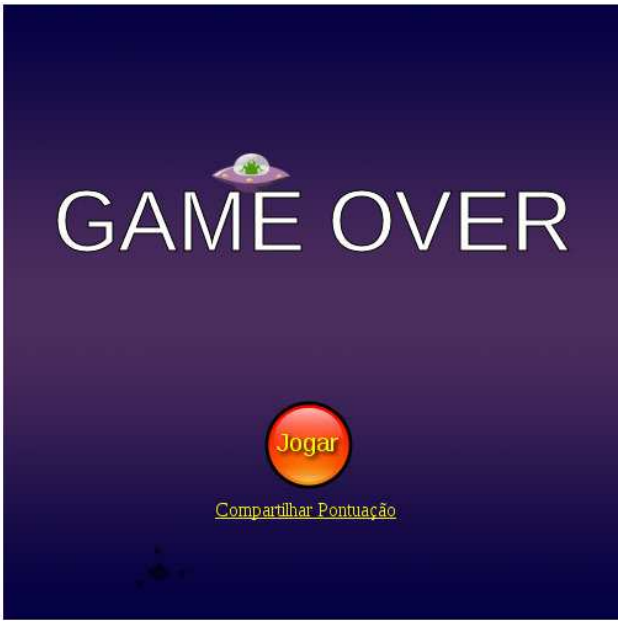


Figura 9.18: Tela de Game Over com link para postar a pontuação no Facebook

Figura 9.19: Popup de compartilhamento do Facebook

Agora seu jogo está pronto para ser divulgado nas redes sociais pelos seus jogadores!

Referências Bibliográficas

[1] David Geary. *Core HTML5 Canvas - Graphics, Animation and Game Development*. 2012.

[2] David Geary. Html5 2d game development: Introducing snail bait. 2012.

[3] W3Schools. Html5 canvas.