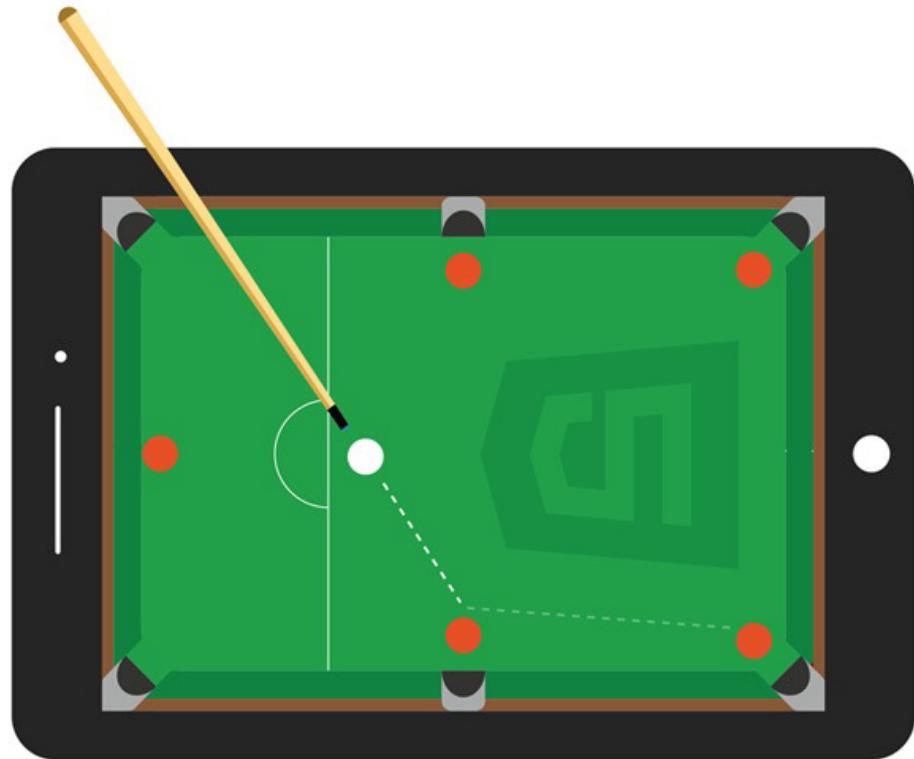


Jogos em HTML5

Explore o mobile e física



Casa do
Código

ÉDERSON CÁSSIO

© Casa do Código

Todos os direitos reservados e protegidos pela Lei nº9.610, de 10/02/1998.

Nenhuma parte deste livro poderá ser reproduzida, nem transmitida, sem autorização prévia por escrito da editora, sejam quais forem os meios: fotográficos, eletrônicos, mecânicos, gravação ou quaisquer outros.

Edição

Adriano Almeida

Vivian Matsui

Revisão

Bianca Hubert

Vivian Matsui

[2017]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

ISBN

Impresso e PDF: 978-85-5519-064-3

EPUB: 978-85-5519-065-0

MOBI: 978-85-5519-066-7

Você pode discutir sobre este livro no Fórum da Casa do Código: <http://forum.casadocodigo.com.br/>.

Caso você deseje submeter alguma errata ou sugestão, acesse <http://erratas.casadocodigo.com.br>.

INTRODUÇÃO

Este livro é continuação de minha obra anterior, *Desenvolva jogos com HTML5 Canvas e JavaScript* (2014). Nela, introduzi a tecnologia Canvas do HTML5 e, principalmente, guiei o leitor pelos conceitos mais básicos envolvidos na criação de jogos em 2D: sprites, loops de animação, input do usuário e outros, sem depender de algum framework específico. Agora, para este novo livro, eu assumo que você já tenha esse conhecimento básico.

O que nós, você e eu, faremos aqui é aprofundar nossos caminhos em algumas coisas mais específicas (leia-se *frameworks*), que são:

- **Apache Cordova:** inicialmente chamado *PhoneGap*, consiste em uma plataforma para a geração de aplicações HTML5 em diversos sistemas operacionais móveis (Android, iOS, Windows Phone etc.). Uma mesma base de código, usando as tecnologias web, rodando como apps em todos esses sistemas!
- **Hammer.js:** biblioteca que detecta gestos comuns em telas sensíveis ao toque, como pinças, rotações e outros, complementando o suporte do HTML5.
- **Box2dWeb:** a fantástica engine de simulação física Box2D, criada em C++ por Erin Catto, foi portada para inúmeras outras linguagens, inclusive, claro, JavaScript. Espero que só este fato já faça valer a pena a leitura deste livro para você, pois é a mesma API (conjunto de classes e métodos) para C++, Java, JavaScript e outros.

Não que eu pretenda que este livro esteja totalmente amarrado ao outro, mas, para facilitar, vou pegar muitos códigos criados anteriormente e reutilizá-los (aprendi no meu jardim de infância da programação que é para isto que serve a Orientação a Objetos). Não se preocupe, caso não tenha lido a primeira obra. Darei uma recapitulada. Se você não a leu, será como mais um frameworkzinho que você está usando.

O livro está dividido em duas partes: *Mobile* e *Física*. Na primeira parte, estão os capítulos:

- **Capítulo Instalação do Apache Cordova:** este framework permite transformar páginas HTML em apps para diversos sistemas operacionais móveis. Aqui explico o que é o Cordova, como funciona e sua instalação passo a passo. Ele se apoia em muitos outros programas que você deverá ter instalados em sua máquina de desenvolvimento.
- **Capítulo Criando o primeiro projeto:** você aprenderá todo o ciclo de criação de um app Cordova, desde a configuração do projeto até a geração do pacote APK do Android assinado para distribuição.
- **Capítulo Eventos touch:** conheceremos a especificação e a API do HTML5 para tratamento de eventos touchscreen. Aprenderemos a verificar as coordenadas, a cruzá-las com algum objeto no Canvas e como fazer para o jogador deslocar sprites arrastando-os com o dedo.
- **Capítulo Incrementando o jogo de nave:** o jogo de guerra espacial que criamos no primeiro livro ganhará duas versões mobile: na primeira, a nave será

controlada tocando botões na tela; na segunda, o jogador terá de inclinar o dispositivo móvel na direção em que ele quer que a nave se mova. Para isso, precisaremos fazer a leitura do acelerômetro do aparelho. Também conheceremos a API de multimídia do Cordova, que permitirá o uso de som em aparelhos cuja *webview* nativa não suporta a API de áudio do HTML5.

- **Capítulo Interações avançadas com Hammer.js:** você aprenderá a usar este framework para detectar gestos de pinça, rotação, deslocamentos e outros. Aqui, criaremos controles para movimentar um taco de bilhar, que será usado em um projeto prático de jogo na segunda parte do livro.

Na segunda parte, é feita uma introdução dos fundamentos do Box2dWeb e, em seguida, guia o leitor na construção de um jogo de bilhar, mostrando como unir a lógica de física com a lógica de negócios e de interação com o jogador:

- **Capítulo Introdução ao Box2dWeb:** conceitos básicos desta engine de física. Criação do mundo físico, corpos, fixtures e gravidade.
- **Capítulo Configuração e movimento de corpos:** crie corpos de diversos formatos, configure sua elasticidade e atrito, aplique forças e impulsos para movimentá-los.
- **Capítulo Iniciando o jogo de bilhar:** construiremos um jogo utilizando a Box2dWeb para controlar os movimentos com física realista. Programaremos os sprites, posicionaremos as bolas na mesa e

simularemos uma tacada, vendo-as se espalharem pelo cenário.

- **Capítulo Interação com o jogador e lógicas de negócio:** incorporamos ao jogo os controles criados com o Hammer.js e executaremos lógicas de acordo com os eventos que ocorrerem, como fazer uma bola encaçapada desaparecer ou retornar ao seu ponto inicial, caso seja a bola branca.
- **Capítulo Simulações internas:** usaremos o Box2dWeb puramente em memória para testar diversas jogadas que o computador pode realizar.
- **Capítulo Finalizando o jogo:** determinaremos qual variedade de bilhar será implementada, e aplicaremos suas regras para selecionar a melhor tacada automática e para arbitrar o andamento do jogo.

O jogo de bilhar pode ser jogado agora mesmo em seu celular ou tablet, no endereço:

<http://edyknopfler.github.io/games-js-2/index.html>



Figura 1: Jogo de bilhar a ser desenvolvido no livro

Procure jogá-lo com o dispositivo móvel na posição paisagem (deitado). Para jogar:

1. Coloque 2 dedos na área circular e gire;
2. Puxe a seta azul para regular a força;
3. Clique em Pow !

Você deve acertar a branca na bola da vez indicada na tela; qual você encaçapa é indiferente. Ganha o jogo quem encaçapar a bola 9.

Todos os códigos, imagens e sons de jogos deste livro podem ser baixados em:

<https://github.com/EdyKnopfler/games-js-2/archive/master.zip>

O jogo de bilhar é disponibilizado sob a licença GNU, não sendo permitido o uso de seu código em aplicações comerciais. Os outros códigos e exercícios estão disponíveis sob a licença Apache, estes, sim, podendo ser usados comercialmente.

Você também precisará dos materiais do meu livro anterior:

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

Continuo à disposição no fórum da Casa do Código, onde já presto suporte aos leitores:

<http://forum.casadocodigo.com.br/>

A instalação do Apache Cordova é um bocado trabalhosa. No primeiro capítulo, tratarei desse assunto exaustivamente. No entanto, reconhecendo a necessidade de uma demonstração

prática para tantos detalhes, gravei uma rápida video-aula como material complementar, na qual demonstro uma instalação do framework em ambiente Windows. Acesse o vídeo em:

<https://www.youtube.com/watch?v=Rf85Y0iNJIE>

Prepare-se! Desejo-lhe uma ótima leitura e ainda mais ótimas experiências futuras com a programação de games!

Sumário

Mobile	1
1 Instalando o Apache Cordova	2
1.1 O que é o Cordova e o que tem a ver com o PhoneGap	2
1.2 Instalando os prerrequisitos	4
1.3 Considerações específicas de sistema operacional	11
1.4 Testando a conexão com o dispositivo	15
2 Criando o primeiro projeto	17
2.1 O projeto Cordova	17
2.2 O arquivo config.xml	18
2.3 Configurando o Android para depuração USB	20
2.4 Rodando o projeto padrão	22
2.5 Um pequeno projeto do zero: animação com o Canvas	24
2.6 Um polyfill para requestAnimationFrame	29
2.7 Gerar pacote APK assinado para o Android	31
2.8 Considerações finais	36
3 Eventos touch	38

Sumário	Casa do Código
3.1 Arrastando um objeto	38
3.2 Controlando a direção	47
3.3 Toque em Canvas responsivo	51
4 Incrementando o jogo de nave	56
4.1 Revisão das classes do jogo	56
4.2 Esqueleto do projeto	58
4.3 API de som do Cordova	66
4.4 Criando um controle direcional	71
4.5 Precisão com múltiplos dedos	82
4.6 API do acelerômetro do Cordova	84
4.7 Adaptando o Canvas ao tamanho da tela	92
5 Interações avançadas com Hammer.js	96
5.1 Conhecendo o Hammer.js	96
5.2 Barra de ajuste de força	103
5.3 Rotacionando um taco de bilhar	112
5.4 Animando a tacada	119
Física	124
6 Introdução ao Box2dWeb	125
6.1 Um primeiro tutorial	127
6.2 Objetos fundamentais	129
6.3 Animação do mundo físico	138
7 Configurações e movimento de corpos	143
7.1 Formas	143
7.2 Movimentando corpos	154

7.3 Propriedades físicas	158
8 Iniciando o jogo de bilhar	164
8.1 Preparação do projeto	165
8.2 Primeiros sprites	170
8.3 As bolas de bilhar	178
8.4 Executando a tacada	185
8.5 Ângulo da trajetória	190
9 Interação com o jogador e lógicas de negócio	195
9.1 Controles da tacada	196
9.2 Executando a tacada no ângulo correto	200
9.3 Posicionando o taco	203
9.4 Tratamento das colisões	205
10 Simulações internas	213
10.1 Realizando várias simulações a partir do início	213
10.2 Salvando e recuperando o estado do mundo físico	217
10.3 Executando a tacada em tempo curto	219
10.4 Tratador de colisões para a simulação	222
11 Finalizando o jogo	227
11.1 Regras do Bola 9	227
11.2 Mostradores	238
Conclusão	242
12 Referências bibliográficas	243

Mobile

CAPÍTULO 1

INSTALANDO O APACHE CORDOVA

Está preparado? Neste capítulo, você tomará contato com o que eu considero uma das ideias mais brilhantes para o desenvolvimento de software. Tudo o que foi feito no primeiro livro, rodando em navegadores desktop, se tornará realidade em dispositivos móveis. Ao longo deste livro, vamos criar versões móveis do jogo de nave, simular partidas de bilhar, além de diversos exercícios com telas touch e física.

Preciso avisar-lhe: para isto se tornar realidade, você terá que instalar várias ferramentas em seu computador. Este primeiro capítulo é dedicado a ensiná-lo como fazer isso. Pode ser que, como desenvolvedor, você já tenha uma ou mais delas. Em todo caso, pegue um café e um prato de biscoitos, ponha uma música relaxante e vamos começar.

1.1 O QUE É O CORDOVA E O QUE TEM A VER COM O PHONEGAP

Se eu lhe perguntar: qual plataforma de software roda em qualquer dispositivo moderno, não importando seu tamanho

físico, velocidade, memória... Java? Talvez, mas existe uma outra: a web. Uma página web pode ser criada para todos os dispositivos imagináveis. Como a tecnologia Canvas faz parte do padrão atual da web (HTML5), podemos dizer o mesmo de jogos criados com ele.

Por que, portanto, não utilizar as tecnologias da web para desenvolvimento móvel?

Em 2008, a Nitobi (hoje adquirida pela Adobe) começou a trabalhar no *PhoneGap*, um sistema que empacota páginas HTML na forma de apps para diversas plataformas móveis. Não há mágica: são, na realidade, aplicações *híbridas*, nas quais o componente nativo de renderização web de cada plataforma é usado para processar o conteúdo em formato web. Sendo apps nativas, podem ser distribuídas nas lojas de aplicativos e rodar localmente.

Em 2011, o projeto foi doado à Fundação Apache, em que passou a se chamar *Cordova*. A Adobe adquiriu a Nitobi logo em seguida e deu foco especial à manutenção do projeto, passando a oferecer um serviço de geração de apps em nuvem, o *Adobe PhoneGap Build* (<https://build.phonegap.com/>), mantendo o nome original. Este serviço, no entanto, é oferecido gratuitamente apenas para projetos de código livre. Por esta razão é que dou preferência, neste livro, à instalação local da biblioteca base.

Mas não se engane: não se trata apenas de um simples renderizador. A *WebView* padrão de cada plataforma é estendida com APIs JavaScript para acesso a diversos recursos dos dispositivos, como telefone, contatos, câmera, acelerômetro, GPS, sistema de arquivos e outros. Enfim, uma plataforma completa

para desenvolvimento de apps, tendo como base — mas não se limitando — ao formato web.

1.2 INSTALANDO OS PRERREQUISITOS

A partir da versão 3.0, o Cordova traz uma interface de linha de comando, a Cordova CLI (*Command Line Interface*). É muito prática de se usar, porém utiliza internamente diversas outras ferramentas conhecidas por desenvolvedores de software.

A seguir, as ferramentas, seus links para download e o devido porquê.

Node.js

<http://nodejs.org/download>

Engine de execução de código JavaScript, sobre a qual a CLI roda. Esta será instalada como um plugin do Node.js. A CLI, em si, é uma aplicação JavaScript do Node.js.

Para Windows e Mac OS X, são fornecidos instaladores. No Linux, você deve extrair o arquivo compactado para o local de sua preferência e configurar o path de seu sistema (mais detalhes na seção *Considerações específicas de sistema operacional*) para enxergar a subpasta `bin`, onde se encontra o executável do Node.js e o utilitário `npm`. Será preciso reiniciar a máquina para o Linux reconhecer novos paths configurados.

	Windows Installer		Macintosh Installer		Source Code
	node-v0.10.29-x86.msi		node-v0.10.29.pkg		node-v0.10.29.tar.gz
Windows Installer (.msi)	32-bit	64-bit			
Windows Binary (.exe)	32-bit	64-bit			
Mac OS X Installer (.pkg)	Universal				
Mac OS X Binaries (.tar.gz)	32-bit	64-bit			
Linux Binaries (.tar.gz)	32-bit	64-bit			
SunOS Binaries (.tar.gz)	32-bit	64-bit			
Source Code	node-v0.10.29.tar.gz				

Figura 1.1: Opções de download do Node.js

O *npm* serve para baixar extensões do Node.js, das quais uma é o Cordova CLI. Para instalá-lo, digite o seguinte em um terminal ou prompt de comando:

```
npm install -g cordova
```

Neste ponto, o Cordova CLI está instalado, mas ainda temos trabalho até poder gerar um app para uma determinada plataforma.

Apache Ant

<http://ant.apache.org/bindownload.cgi> (sim, é "bindownload" tudo junto)

Ferramenta de automação bem conhecida por desenvolvedores Java, e é requerida pela CLI na hora de gerar projetos Android nativos. Se esta ferramenta não estiver instalada, você verá mensagens de erro na hora de adicionar a plataforma Android ao seu projeto Cordova.

Vá ao endereço indicado e role a página para encontrar os links da versão atual (*Current Release of Ant*). Você novamente terá de extrair os arquivos e configurar a subpasta bin no path do seu sistema.

Current Release of Ant

Currently, Apache Ant 1.9.4 is the best available version, see the [release notes](#).

Note

Ant 1.9.4 was released on 05-May-2014 and may not be available on all mirrors for a few days.

Tar files may require gnu tar to extract

Tar files in the distribution contain long file names, and may require gnu tar to do the extraction.

- .zip archive: [apache-ant-1.9.4-bin.zip](#) [PGP] [SHA1] [SHA512] [MD5]
- .tar.gz archive: [apache-ant-1.9.4-bin.tar.gz](#) [PGP] [SHA1] [SHA512] [MD5]
- .tar.bz2 archive: [apache-ant-1.9.4-bin.tar.bz2](#) [PGP] [SHA1] [SHA512] [MD5]

Figura 1.2: Opções de download do Apache Ant

Git

<http://git-scm.com/downloads>

A partir da versão 3.0, as diferentes APIs do Cordova foram quebradas em plugins, que devem ser baixados e instalados nos devidos projetos. Quem se encarrega desses downloads é o Git,

conhecida ferramenta de gerenciamento de repositórios (como o GitHub). O Cordova também cuida de chamá-lo no momento da instalação de um plugin.

SDKs

- **Android (requer Java SDK):** primeiro <http://www.oracle.com/technetwork/pt/java/java-se/downloads>, depois <http://developer.android.com/sdk/>
- **iOS:** <http://developer.apple.com/programs/ios/>
- **Windows Phone:** <http://developer.windowsphone.com/>

Software Development Kits das diversas plataformas, usados para compilar os componentes nativos. Estes são só alguns exemplos, veja a lista completa em <http://goo.gl/NkIfhK>. Vou dar o exemplo da configuração de um ambiente Android, mas quero que você tenha em mente que este link traz detalhes sobre a configuração em cada plataforma suportada pelo Cordova.

Ou seja, não há nenhuma mágica: se o Cordova é capaz de gerar apps para diversas plataformas, é porque foi criada a comunicação com cada um dos SDKs nativos. Você pode instalar todos, ou somente aqueles que deseja usar. Neste livro, focarei no Android.

O Android SDK requer o SDK da plataforma Java (Oracle JDK) para funcionar. Programadores Java já o têm instalado no computador; se não é o seu caso, dirija-se à primeira URL indicada

e role a tela para encontrar o JDK 7. Infelizmente, enquanto escrevo este livro, a plataforma Android ainda não suporta o Java 8.

The screenshot shows the Java SE 7u60 download page. At the top, it says "Java SE 7u60" and "This update release contains several enhancements and changes including the following:" followed by a bulleted list: "Java Mission Control 5.3" and "New Features and Changes". Below this is a "Learn more" link. To the right, there are three download buttons: "JDK DOWNLOAD", "Server JRE DOWNLOAD", and "JRE DOWNLOAD". A red arrow points from the left margin towards the "JDK DOWNLOAD" button. On the left side, there is a vertical list of links: "Installation Instructions", "Release Notes", "Oracle License", "Java SE Products", "Third Party Licenses", "Certified System Configurations", "Readme Files" (with sub-links "JDK Readme" and "JRE Readme").

Figura 1.3: Download do JDK 7

Após instalado o JDK, você deve configurar a variável de ambiente `JAVA_HOME` (seção *Considerações específicas de sistema operacional* adiante), apontando para a pasta onde ele está instalado, por exemplo: `C:\Program Files\Java\jdk1.7.0`.

O SDK do Android em si pode ser baixado de duas formas: junto com o Android Studio (IDE completa para desenvolvimento nativo) e somente ferramentas do SDK. Para desenvolvimento com o Cordova CLI, usaremos apenas as ferramentas *stand-alone* (se você já desenvolve para Android com o Android Studio ou o antigo ADT, não precisa baixar, pode usar a SDK integrada). Procure os links no tópico `SDK Tools Only`:

SDK Tools Only

If you prefer to use a different IDE or run the tools from the command line or with build scripts, you can instead download the stand-alone Android SDK Tools. These packages provide the basic SDK tools for app development, without an IDE. Also see the [SDK tools release notes](#).

Platform	Package	Size	SHA-1 Checksum
Windows	installer_r24.0.1-windows.exe (Recommended)	92180986 bytes	505d7a95647bcc194b7aa707854422d9c7288d5
	android-sdk_r24.0.1-windows.zip	140743633 bytes	cc49974e8bfcc865ffe0d887e9a74cf52085c632
Mac OS X	android-sdk_r24.0.1-macosx.zip	88535741 bytes	7097c09c72645d7ad33c81a37b1a1363a9df2a54
Linux	android-sdk_r24.0.1-linux.tgz	141304203 bytes	fb46b9afa04e09d3c33fa9bfee5c99e9ec6a9523

Figura 1.4: Download do Android SDK

Após instalar o Android SDK, configure no path as subpastas `tools` e `platform-tools` (esta última ainda a ser instalada). Inicie um terminal ou prompt de comando (no Windows, como administrador) e digite:

```
android
```

Será aberto o *SDK Manager*, no qual você terá de fazer o download dos itens marcados nas figuras seguintes: *Platform-tools*, *Build-tools* e *SDK Platform* da API 21 do Android:

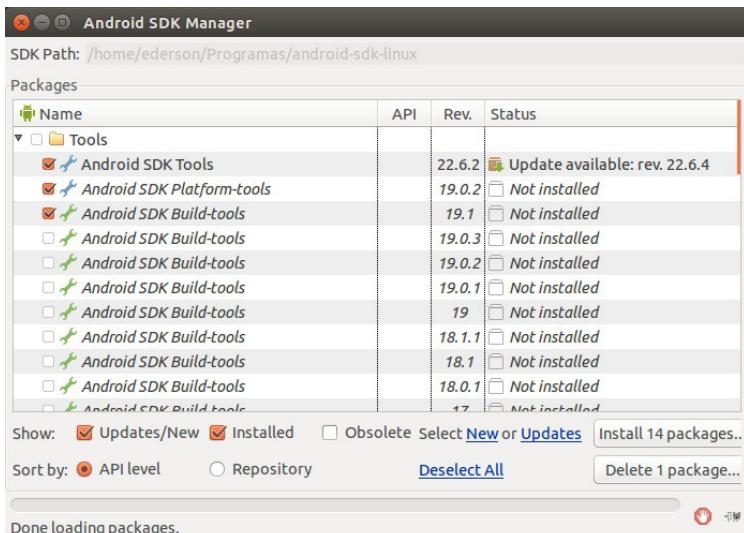


Figura 1.5: Ferramentas de build do Android SDK: tools e platform-tools

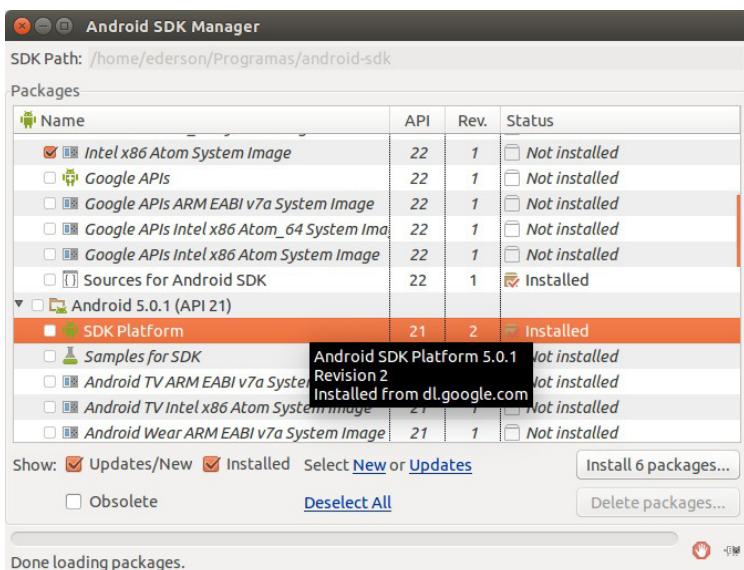


Figura 1.6: SDK Platform da API 21 do Android

APIs ANDROID

As versões lançadas do Cordova tentam acompanhar as APIs Android. No momento em que finalizava a primeira edição deste livro, o Cordova dava suporte até a API 19. Caso você obtenha erros de versão da API ao adicionar a plataforma Android a seus projetos (capítulo *Criando o primeiro projeto*), você pode instalar sucessivamente as novas versões até encontrar uma com suporte.

EMULADORES VS. DISPOSITIVOS REAIS

O Android SDK possui um emulador. Para usá-lo, você deve instalar pelo menos uma *System Image* de uma determinada API no SDK Manager. No entanto, o emulador é **muito** lento. Pode ter desempenho aceitável para testar apps comuns, em máquinas com processadores rápidos e bastante memória.

No caso de jogos, procure usar um dispositivo real para testá-los.

1.3 CONSIDERAÇÕES ESPECÍFICAS DE SISTEMA OPERACIONAL

Windows

Para testar jogos em dispositivos conectados no Windows, é preciso instalar um driver USB. Há uma lista em <http://developer.android.com/tools/extras/oem-usb.html#Drivers>.

Após instalar o driver, procure reiniciar a máquina.

Muito provavelmente você sabe como configurar o path do sistema operacional, ou criar a variável de ambiente JAVA_HOME , no entanto, deixo aqui registrados os passos:

- Clique com o botão direito em Computador e escolha Propriedades .
- À esquerda da tela, escolha Configurações Avançadas do Sistema .
- Clique no botão Variáveis de Ambiente .

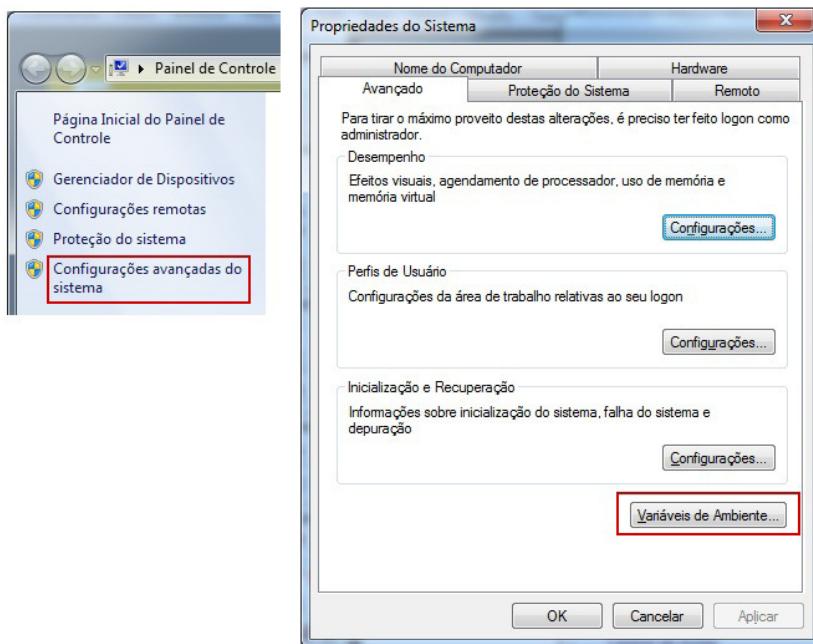


Figura 1.7: Entrando nas variáveis de ambiente do Windows

Para acrescentar uma pasta ao path:

- Procure em uma das listas a variável PATH e clique em Editar .
- Acrescente um ponto e vírgula ao final dos caminhos e insira o novo.

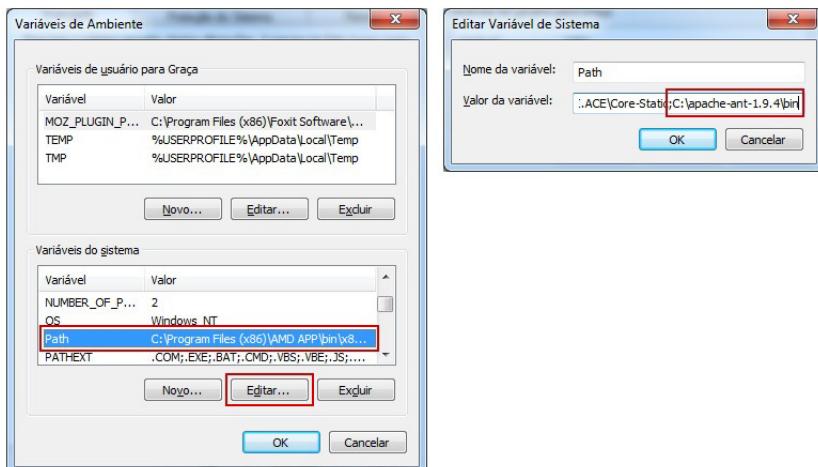


Figura 1.8: Acrescentando uma pasta no path (Apache Ant, no exemplo)

Linux

No Linux, é preciso configurar uma permissão de acesso via USB para aparelhos de determinado fabricante:

1. Conecte seu dispositivo ao computador pelo cabo USB.
2. Abra o terminal e digite o comando `lsusb`. Procure na saída do comando o seu dispositivo e anote o primeiro código hexadecimal. Ele corresponde ao código do fabricante:

```
ederson@Ederson-Lenovo:~$ lsusb
Bus 002 Device 002: ID 0bda:0139 Realtek Semiconductor Corp. RTS5139 Card Reader Controller
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 001 Device 005: ID 22b8:2e76 Motorola PCS
Bus 001 Device 002: ID 04f2:b272 Chicony Electronics Co., Ltd Lenovo EasyCamera
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 003 Device 002: ID 04f3:0235 Elan Microelectronics Corp.
Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
ederson@Ederson-Lenovo:~$
```

Figura 1.9: Encontrando o código do fabricante (o da Motorola é 22b8)

3. Edite como administrador o arquivo:

```
sudo gedit /etc/udev/rules.d/51-android.rules
```

4. Insira a linha (pode ser ao final do arquivo, caso ele já exista):

```
SUBSYSTEM=="usb", ATTRS{idVendor}=="22b8", MODE="0666"
```

Isto define uma regra dando ao sistema a permissão de acesso aos dispositivos daquele fabricante.

5. Mude a permissão do arquivo para 644 :

```
sudo chmod 644 /etc/udev/rules.d/51-android.rules
```

No caso, o 6 representa as permissões de leitura e gravação (4 + 2) ao usuário corrente (que é o administrador, devido ao uso do sudo); o 4, a permissão de leitura para o grupo a que o usuário pertence; e o outro 4, a permissão de leitura aos outros usuários.

6. Mude o proprietário do arquivo para o administrador (root):

```
sudo chown root /etc/udev/rules.d/51-android.rules
```

As variáveis de ambiente podem ser configuradas editando-se como administrador o seguinte arquivo (e reiniciando a máquina depois):

```
sudo gedit /etc/environment
```

Ao contrário do Windows, os diretórios são separados por dois pontos:

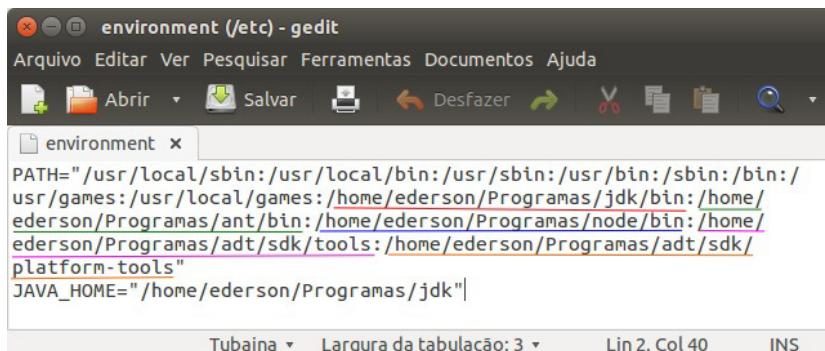
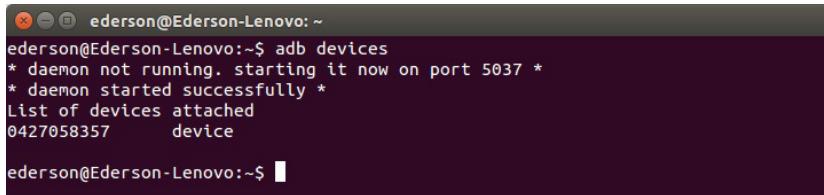


Figura 1.10: Configuração de path e JAVA_HOME no Linux. Cada path configurado manualmente é destacado com uma cor, para facilitar a visualização

Após realizar todas as edições, reinicie a máquina.

1.4 TESTANDO A CONEXÃO COM O DISPOSITIVO

Em qualquer caso (Windows ou Linux), podemos testar se o SDK consegue "conversar" com o celular ou tablet. Abra um prompt de comando ou terminal, e dê o comando `adb devices`. Este é um comando do Android SDK que lista os aparelhos conectados:



```
ederson@Ederson-Lenovo:~$ adb devices
* daemon not running. starting it now on port 5037 *
* daemon started successfully *
List of devices attached
0427058357    device
ederson@Ederson-Lenovo:~$
```

Figura 1.11: Dispositivos conectados

Se aparecerem vários pontos de interrogação ou a indicação "*no permissions*", confira os passos citados para seu sistema operacional. Mais detalhes podem ser obtidos em <http://developer.android.com/tools/device.html>. E em <http://goo.gl/42Xila>, podemos ver uma descrição detalhada de como configurar o acesso ao dispositivo móvel no Linux.

Se seu dispositivo aparece como "device" após o comando `adb devices`, abra uma cerveja! Estamos prontos para finalmente começar a usar o Apache Cordova. Já temos as ferramentas necessárias para trazer para o mundo mobile todas as experiências que somos capazes de criar com o HTML5. Para dar continuação ao processo, já no próximo capítulo, você aprenderá a criar projetos e rodá-los em seu celular ou tablet, e poderá ver animações em Canvas funcionando. Em breve, nossos esforços se transformarão em jogos reais e divertidos.

SUGESTÃO DE LEITURA

Apache Cordova 3 Programming, de John M. Wargo (2013).

CAPÍTULO 2

CRIANDO O PRIMEIRO PROJETO

É hora de tirar a prova dos nove: vamos usar a interface de linha de comando (CLI) do Cordova para criar um pequeno projeto HTML5. Se algo der errado, certifique-se de que todas as ferramentas instaladas estejam configuradas no path do sistema, conforme descrito exaustivamente no capítulo anterior.

2.1 O PROJETO CORDOVA

No prompt de comando do Windows, ou no terminal do Linux, navegue até a pasta onde você pretende salvar os códigos deste livro. Estando nessa pasta, dê o comando:

```
cordova create primeiro
```

O Cordova cria a pasta `primeiro` dentro da atual. Vamos abri-la:

```
cd primeiro
```

Os próximos comandos atuarão no projeto configurado nesta pasta. Vamos adicionar a plataforma Android:

```
cordova platform add android
```

Vale a pena darmos uma olhada na estrutura de pastas do projeto. Destaque para o arquivo `config.xml`, a ser descrito na próxima seção, para a pasta `www`, onde fica o projeto web, e `platforms`, onde o Cordova cria projetos nativos para os SDKs das plataformas adicionadas:



Figura 2.1: Estrutura de pastas de um projeto Cordova

Nosso trabalho será praticamente todo feito na pasta `www`, pois é lá que deverão ser salvos os arquivos HTML, CSS e JavaScript de nossos apps e jogos. Recomenda-se não editar diretamente o conteúdo da pasta `platforms`, pois este é sempre regerado automaticamente a cada execução do projeto. Nada impede, porém, que copiemos o projeto Android nativo para outro local e o abramos no ADT, por exemplo.

2.2 O ARQUIVO CONFIG.XML

Neste arquivo, configuramos algumas propriedades do projeto usadas na geração da app nativa. Destaque para o atributo `id` da tag `<widget>`, onde definimos o *pacote*, nome que identificará nosso app de forma exclusiva. Editei o arquivo como a seguir; sinta-se livre para colocar os seus próprios dados:

```
<?xml version='1.0' encoding='utf-8'?>
```

```
<widget id="br.com.casadocodigo.livrojogoshtml2.primeiro"
    version="0.0.1"
    xmlns="http://www.w3.org/ns/widgets"
    xmlns:cdv="http://cordova.apache.org/ns/1.0">

    <name>Primeiro Projeto Cordova</name>

    <description>
        Primeiro teste de empacotamento de páginas HTML
        em apps móveis.
    </description>

    <author email="ederson_cassio@yahoo.com.br"
        href="http://gamecursos.com.br">
        Éderson Cássio
    </author>

    <content src="index.html" />
    <access origin="*" />
</widget>
```

Uma vez publicado o app na Play Store, o nome do pacote não deverá mais mudar. A convenção, já conhecida por programadores Java, é usar a URL reversa, colocando ao final o nome que identifica o app ou jogo. Por exemplo, `br.com.casadocodigo.livrojogoshtml2`, em que `br.com.casadocodigo` corresponderia ao seu URL na web, e `livrojogoshtml2` identifica este livro. Vamos continuar colocando mais pontos e novos nomes, para identificar cada joguinho criado aqui.

ÍCONE DO APLICATIVO

No config.xml , você pode definir o ícone de um aplicativo pela tag <icon> :

```
<icon src="res/icon.png" />
```

Mais detalhes podem ser obtidos em <http://goo.gl/Q6Nmz3>.

2.3 CONFIGURANDO O ANDROID PARA DEPURAÇÃO USB

Se você já desenvolve para Android, sinta-se livre para pular esta parte.

Por razões de segurança, não se pode sair copiando aplicativos para os aparelhos. Para desenvolvimento, temos de habilitar a **Depuração USB**:

HABILITANDO AS OPÇÕES DE DESENVOLVEDOR NO ANDROID 4.2 E POSTERIORES

A partir do Android 4.2, a opção Desenvolvedor das Configurações está oculta. Para fazê-la aparecer, entre em Sobre o telefone , nas configurações, e toque sete vezes a opção Número da versão (figura a seguir).

- Procure nas configurações de seu aparelho a opção

Desenvolvedor ou Programador , conforme o modelo.

- Habilite o modo desenvolvedor e a depuração USB (figura *Habilitando a depuração USB*).

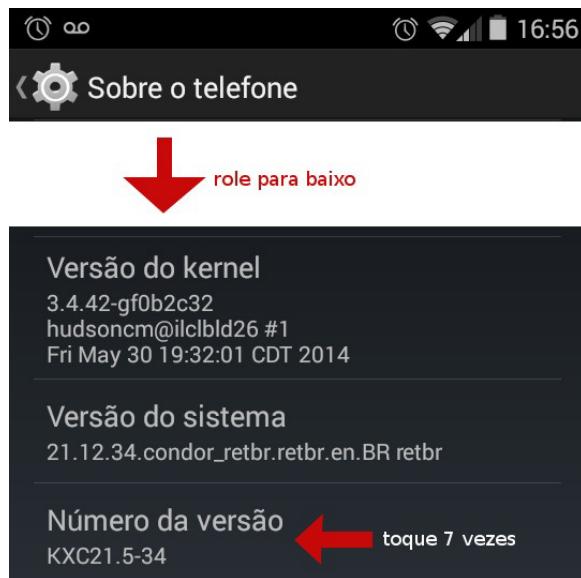


Figura 2.2: Habilitando as opções de desenvolvedor a partir do Android 4.2

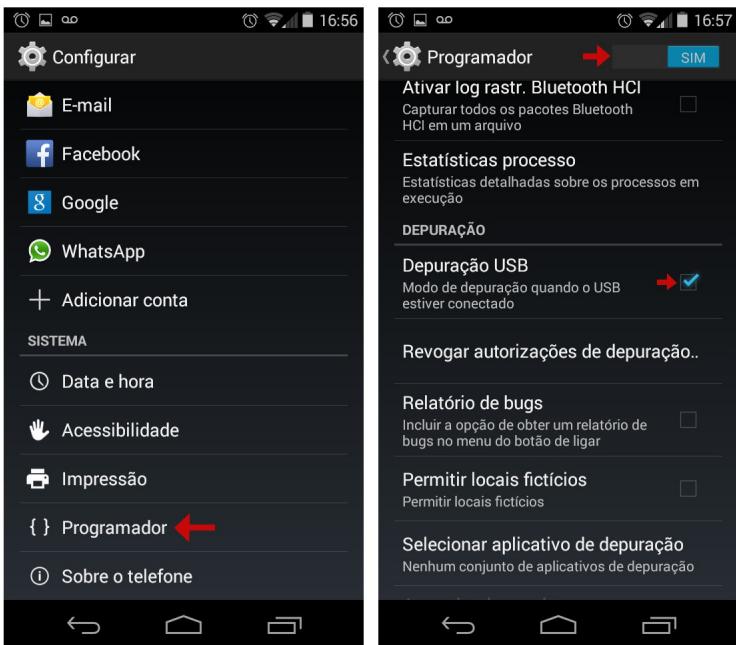


Figura 2.3: Habilitando a depuração USB

2.4 RODANDO O PROJETO PADRÃO

O Cordova já inseriu alguns arquivos na pasta `www`. Vamos rodar o projeto para nos certificarmos de que está tudo ok.

- Conecte o aparelho. A partir do Android 4.2.2, a cada computador diferente conectado, aparecerá no display um pedido de autorização para uma chave RSA específica para aquele computador:

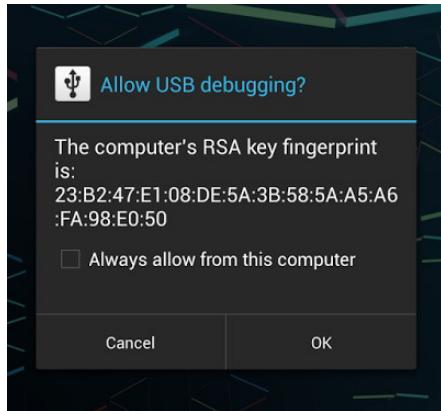


Figura 2.4: Confirmação para depuração USB a partir do Android 4.4.2

- Abra o prompt ou terminal, e entre na pasta de seu projeto:

```
cd primeiro
```

- Mande rodar o projeto:

```
cordova run android
```

O primeiro build leva alguns minutos. Se tudo estiver ok e o Android SDK conseguir comunicar-se com o dispositivo, você verá no terminal a mensagem `LAUNCH SUCCESS` e o projeto padrão do Cordova rodando no display. Nesta altura, como foram muitas configurações, pode ser que ocorram erros. Certifique-se de que todas as ferramentas utilizadas pelo Cordova listadas na seção *Instalando os pré-requisitos* (capítulo anterior) tenham sido instaladas.

Reveja as considerações sobre os sistemas operacionais (seção *Considerações específicas de sistema operacional*) e garanta que os

paths estejam devidamente configurados. Pode ser que, após alguma instalação ou configuração, seja preciso reiniciar o computador.

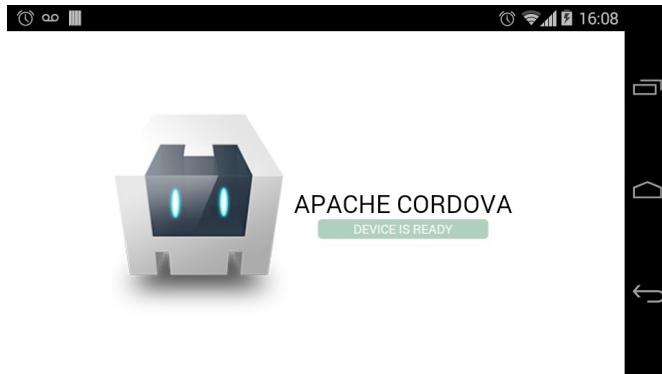


Figura 2.5: Projeto padrão do Cordova rodando no dispositivo móvel

2.5 UM PEQUENO PROJETO DO ZERO: ANIMAÇÃO COM O CANVAS

A página inicial

Vamos apagar os arquivos presentes na pasta `www` e começar recriando o `index.html`. Este representa a página inicial do projeto. Note as tags `<meta name="viewport" ...>` e `<script src="cordova.js">`:

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <title>Primeiro Projeto</title>
    <meta name="viewport"
        content="width=device-width, height=device-height,
        user-scalable=no, initial-scale=1,
```

```

        maximum-scale=1, minimum-scale=1">
<script src="cordova.js"></script>
<script>
    // Aqui vai nosso tratamento do Canvas
</script>
</head>

<body>
    <canvas id="primeiro" width="320" height="240"></canvas>
</body>

</head>

```

O *viewport* é conhecido de quem já trabalha com webdesign responsivo. Trata-se do tamanho do conteúdo. Se não estiver definido, o dispositivo trata a página como sendo para desktop, com um viewport maior que 900 pixels (o valor padrão varia entre os dispositivos), como uma forma de lidar com páginas não preparadas para as telas pequenas. É por isso que vemos muitos sites totalmente comprimidos na minúscula tela e temos que fazer a pinça (zoom) a todo momento:



Figura 2.6: Um site para desktop aparece minúsculo no dispositivo móvel (imagem editada para suprimir anúncios)

Em nosso caso, os atributos `width=device-width` e

`height=device-height` estão definindo as dimensões do viewport no tamanho reportado pelo aparelho. Os atributos usados são:

- `width` : largura do viewport;
- `height` : altura do viewport;
- `user-scalable` : define se o usuário pode dar zoom. Para jogos, em muitos casos é interessante setar como `no` (sem zoom), mas isso dependerá das características do jogo;
- `initial-scale` : zoom inicial. O valor `1` define o zoom "normal" (100%). Valores abaixo (como `0.5`) reduzem a visão da página, e valores acima (por exemplo, `2`) ampliam;
- `maximum-scale` : zoom máximo permitido;
- `minimum-scale` : zoom mínimo permitido.

Definimos aqui características bastante restritivas para o zoom. Para sites, isto é totalmente contra os bons princípios de usabilidade, mas aqui estamos falando de jogos (ou apps) em geral. Nesses casos, permitir ou não o zoom, em que quantidade e em que situações, são decisões do projeto.

Temos também o script `cordova.js`, cujo arquivo... não existe no projeto! Ele estará acessível somente no webview do Cordova. Enquanto não usarmos nenhuma API específica do Cordova, poderemos até mesmo testar nosso jogo no browser do desktop e ignorar este arquivo. Porém, a partir do momento em que chamamos APIs do Cordova, estas só rodarão em seu ambiente, e só poderemos testá-las com um aparelho móvel ou emulador.

Iniciando a API

Para aguardar a página carregar no navegador, nós fazemos o seguinte script (pode incluí-lo na tag `<script>` vazia):

```
window.onload = function() {  
    // Nosso código de inicialização  
}
```

No Cordova, ainda temos de aguardar o carregamento de suas APIs. O ambiente nos fornece o evento `deviceready` :

```
window.onload = function() {  
    document.addEventListener('deviceready', iniciar);  
}  
  
function iniciar() {  
    // Nosso código de inicialização  
}
```

Mas não temos esse evento no browser, caso queiramos testar. Bastaria chamar a função `iniciar` diretamente no evento `onload` :

```
//document.addEventListener('deviceready', iniciar);  
iniciar();
```

Quando não estamos usando APIs móveis específicas, isso pode nos permitir depurar melhor o jogo (através do console do Google Chrome ou do plugin Firebug do Firefox, por exemplo). Lógicas de Canvas ou de engines físicas (como a Box2dWeb, que aprenderemos na Parte 2 do livro) não necessitam necessariamente de dispositivos móveis para rodar e podem ser depuradas em ambiente desktop. Entradas simples via teclado ou mouse podem ser usadas para simular eventos chave a serem testados.

E vamos para a inicialização: obtenção do Canvas e seu

contexto gráfico, disparo do loop de animação... Façamos algo básico e conhecido por enquanto, o complexo problema matemático da *bolinha quicando*:

```
var canvas, context;
var x, y, raio = 10;
var vX = 10, vY = -20; // Velocidades

window.onload = function() {
    //document.addEventListener('deviceready', iniciar);
    iniciar();
}

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

    // Começar no meio da tela
    x = canvas.width / 2;
    y = canvas.height / 2;

    requestAnimationFrame(animar);
}
```

Estamos disparando um loop de animação com a função `animar`, portanto, vamos criá-la:

```
function animar() {
    // Não deixar passar das bordas
    // Como x e y são o centro, descontamos o raio
    // Se passou do limite, invertemos o sentido da velocidade
    if (x < raio || x > canvas.width - raio)
        vX *= -1;

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

    x += vX;
    y += vY;

    context.clearRect(0, 0, canvas.width, canvas.height);

    context.beginPath();
```

```
    context.arc(x, y, raio, 0, Math.PI*2);
    context.fill();

    requestAnimationFrame(animar);
}
```

Se abrirmos a página no browser, já veremos a bolinha quicando. No celular ou tablet, pode ser que sim, pode ser que não. Por quê? Veja na próxima seção.

2.6 UM POLYFILL PARA REQUESTANIMATIONFRAME

O suporte para o `requestAnimationFrame` em dispositivos móveis é bem recente. O browser do Android somente o aceita a partir da versão 4.4 (<http://caniuse.com/requestanimationframe>)! Um jogo hospedado na web, acessado pelo Chrome ou Firefox devidamente atualizados, pode rodar normalmente. As aplicações Cordova, no entanto, usam o webview padrão do sistema operacional. No Safari do iOS, o suporte já existia há algumas versões atrás, mas também é recente.

Nesses casos, a solução é usar um *polyfill*: uma implementação JavaScript que tenta emular o comportamento de uma funcionalidade recente e ainda não completamente suportada. No caso do `requestAnimationFrame`, o velho `setTimeout` pode ser usado para construir um polyfill.

Um exemplo simples poderia ser como o mostrado a seguir. Primeiro, checamos a existência por ordem de prioridade da especificação padrão `requestAnimationFrame`, ou das versões prefixadas com `webkit`, `moz` ou `ms` (específicas de navegador, de quando não tínhamos uma especificação definida). Usamos

para isso o *ou lógico* (||), que fará retornar a primeira referência que for diferente de null . Caso nenhuma delas exista (todas null), retornamos uma função construída usando o setTimeout , que tenta simular 60 frames por segundo:

```
window.requestAnimationFrame =
  // Especificação padrão
  window.requestAnimationFrame      ||
  // Chrome, Safari e outros
  window.webkitRequestAnimationFrame ||
  // Firefox
  window.mozRequestAnimationFrame   ||
  // IE
  window.msRequestAnimationFrame    ||
  // Função substituta
  function(callback) {
    window.setTimeout(callback, 1000 / 60);
 };
```

Dividindo 1 segundo (1000ms) por 60, nosso polyfill tentará chegar perto dos 60fps, tanto quanto a precisão do setTimeout permitir. Isso foi só uma forma mais compacta de se fazer:

```
// Exemplo teórico
var funcao;

if (window.requestAnimationFrame)
  funcao = window.requestAnimationFrame;
else if ( ... ) // etc.

window.requestAnimationFrame = funcao;
```

Erik Möller, alemão vice-diretor da Wikimedia Foundation e desenvolvedor, criou um polyfill bastante robusto, que pode ser obtido em <https://gist.github.com/paulirish/1579671>. Este polyfill mede o tempo entre o frame anterior e o atual, e desconta o atraso para a próxima chamada. Salve o código do polyfill na pasta www e faça o link com a página na seção <head> :

```
<script src="rAF.js"></script>
```

E troque a chamada à função `iniciar`:

```
document.addEventListener('deviceready', iniciar);  
//iniciar();
```

Agora você pode fazer novamente:

```
cordova run android
```

A animação funcionará em qualquer aparelho Android.

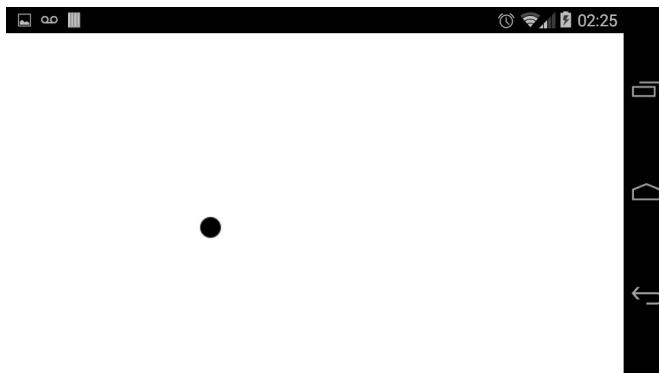


Figura 2.7: Animação com Canvas e requestAnimationFrame na forma de app Android

2.7 GERAR PACOTE APK ASSINADO PARA O ANDROID

Hora de gerar os arquivos `.apk`, os pacotes de aplicativos Android. Estes pacotes devem ser assinados para que possam ser publicados na PlayStore, ou instalados diretamente nos aparelhos.

Gerando uma chave de assinatura

Primeiro devemos gerar uma chave de assinatura, usando o

comando `keytool`, presente na JDK. Esta chave será salva em um arquivo `.keystore`, que pode conter uma ou mais chaves. Siga os passos:

- Abra o terminal ou prompt em uma pasta separada do projeto (pode ser na pasta padrão do usuário, onde o prompt começa).
- Dê o comando para criar a chave. Aqui quebrei em várias linhas para facilitar, mas você deverá digitar tudo em uma única linha e substituir "ederson-cassio" por seu nome ou empresa, naturalmente. Neste caso, a chave possuirá o *alias* (nome) `ederson_cassio`, será armazenada no arquivo `ederson-cassio.keystore` e sua validade será de 10000 dias (aproximadamente 27 anos, mais que o suficiente):

```
keytool
-genkey -v
-keystore ederson-cassio.keystore
-alias ederson_cassio
-keyalg RSA
-keysize 2048
-validity 10000
```

- Informe a senha da keystore (será pedida confirmação, em caso de nova keystore).
- Informe os dados de identificação pedidos (seu nome, empresa, localidade). A chave será gerada em seguida.
- Ao final, pede-se a senha da nova chave. Você pode apenas dar `Enter` se quiser usar a mesma senha da keystore.

Gerando o APK para distribuição

Agora vá para a pasta do projeto e dê o comando:

```
cordova build android --release
```

Será gerado o arquivo `PrimeiroProjetoCordova-release-unsigned.apk`, na subpasta `platforms/android/ant-build`. Também estão ali os APKs de *debug* gerados anteriormente, que só podem ser usados nos aparelhos com depuração USB ativada.

Assinando o APK

Para facilitar, mova o arquivo `PrimeiroProjetoCordova-release-unsigned.apk` para a pasta onde está a keystore. Você poderá renomeá-lo para algo mais significativo, como `MeuJogo.apk`.

De volta ao prompt, na pasta inicial, podemos assinar o arquivo pelo comando a seguir. Perceba que os três últimos parâmetros referem-se ao arquivo `.keystore`, ao arquivo `.apk` e ao *alias* armazenado:

```
jarsigner  
-verbose -sigalg SHA1withRSA -digestalg SHA1  
-keystore ederson-cassio.keystore  
MeuJogo.apk  
ederson_cassio
```

Basta informar a senha da keystore e o arquivo será assinado.

Instalando o APK diretamente no aparelho

Para poder instalar um aplicativo diretamente do APK, é necessário configurar o Android para aceitar apps de fontes desconhecidas:

- Entre nas Configurações e, em seguida, em Segurança .
- Role até encontrar a opção Fontes desconhecidas e marque-a:

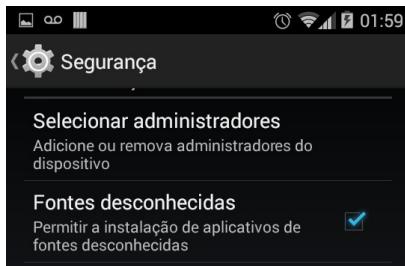


Figura 2.8: Configurando o Android para aceitar APKs estranhos

O Android dará um aviso de segurança. Recomendo que você não o ignore; após instalar o aplicativo, procure desativar essa opção.

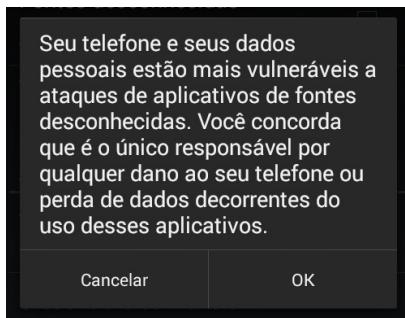


Figura 2.9: Não diga que não foi avisado...

- Copie o arquivo .apk para o aparelho e localize-o. Pode ser que você tenha de instalar

um gerenciador de arquivos. No meu celular, estou usando o *ES File Explorer*, que é gratuito:

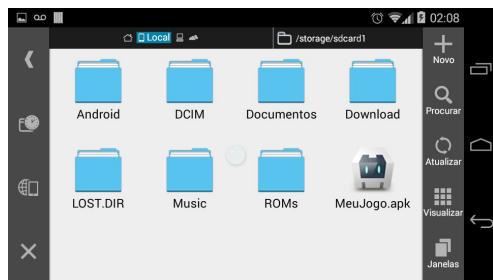


Figura 2.10: Procurando o arquivo APK

PERMISSÕES DO ANDROID

Quem é usuário do Android sabe que, para instalar um aplicativo ou jogo, este requer um certo número de permissões do sistema operacional. O Cordova, por padrão, coloca no projeto um pedido de permissão para acesso à internet. Se seu jogo utilizar apenas recursos locais, podemos remover esta permissão:

- Na subpasta `platforms/android` , abra o arquivo `AndroidManifest.xml`
- Remova a seguinte tag:

```
<uses-permission  
    android:name="android.permission.INTERNET" />
```

Muito cuidado, porém: a instalação de plugins do Cordova, que aprenderemos na seção *API de som do Cordova*, poderá adicionar pedidos de diversas permissões requeridas pelas APIs. Não remova os novos requerimentos que aparecerem, ou as APIs não funcionarão. Você pode colocar de volta a permissão para internet caso tenha constatado alguma falha.

2.8 CONSIDERAÇÕES FINAIS

É muita coisa técnica! Mas isso tudo é necessário para termos nossos jogos na forma de apps Android. Dominando todo este ciclo de desenvolvimento, já podemos nos concentrar em criar coisas mais divertidas. No próximo capítulo, brincaremos com a interação por toque.

O que ocorre é que o uso do HTML5 é uma tendência recente, porém em crescimento, para o desenvolvimento de aplicativos em diversas plataformas (como é o caso do Windows 8 e do Firefox OS). Ainda não há IDEs e ferramentas de produtividade. Aplicações híbridas (como as geradas pelo Cordova) tendem a ser mais lentas que as nativas, em especial nos aparelhos mais simples, mas a potência deles só tende a aumentar. A ideia de usar o HTML5 para alcançar qualquer plataforma está se firmando.

DOCUMENTAÇÃO DO CORDOVA

Para aprofundar seus estudos, deixo-lhe o link para a documentação do projeto:

<http://cordova.apache.org/docs/en/3.5.0/>

Lá você poderá aprender sobre as opções da CLI, configurações do config.xml , APIs, plugins e muito mais.

CAPÍTULO 3

EVENTOS TOUCH

Vamos tratar agora da especificação do HTML5 para eventos touch. Existem frameworks para auxiliar no tratamento de gestos complexos e usuais (mais adiante, no capítulo *Interações avançadas com Hammer.js*, abordarei o *Hammer.js*), que são bastante trabalhosos de tratar. Ainda assim, devemos saber como tratar, pelo menos, as interações básicas com o HTML5 "cru".

A especificação define os seguintes eventos:

- `touchstart` : início do toque;
- `touchmove` : usuário move o dedo na tela a partir do objeto que recebeu o evento;
- `touchleave` : o dedo abandona a área do objeto;
- `touchenter` : o dedo entra de volta na área do objeto;
- `touchend` : fim do toque (usuário solta o dedo);
- `touchcancel` : o cancelamento do toque ocorre se a área da página sair do foco de interação (por exemplo, se surgir uma caixa de alerta).

3.1 ARRASTANDO UM OBJETO

É a base de interação em múltiplos jogos para telas touch (quem nunca jogou *Angry Birds* ou *Pou?*). Com o auxílio dos

eventos `touchstart`, `touchmove` e `touchend`, podemos obter as coordenadas da tela onde o toque ocorreu, verificar se o objeto se encontra por ali e movê-lo.

Preparei uma pequena *spritesheet* para controlarmos com o dedo. Em <http://www.iconarchive.com/>, e muitos outros sites, você pode obter diversos ícones, a partir dos quais você poderá montar spritesheets usando um editor de imagens:

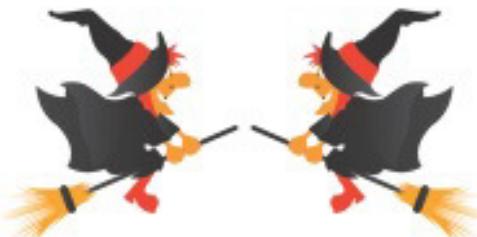


Figura 3.1: É um pássaro? É um avião? Use sua imaginação!

O esqueleto

Crie um novo projeto Cordova:

```
cordova create touch
```

Adicione a plataforma Android:

```
cd touch
cordova platform add android
```

Coloque o seguinte esqueleto no `index.html`:

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <title>Arrastando Objeto Via Touch</title>
```

```

<meta name="viewport" content="width=device-width,
      height=device-height, user-scalable=no,
      initial-scale=1, maximum-scale=1, minimum-scale=1">
<script src="cordova.js"></script>
<script src="rAF.js"></script>
<script>
    // Código do app
</script>
</head>

<body>
    <canvas id="touch" width="320" height="240"></canvas>
</body>

</head>

```

Façamos a inicialização básica discutida no capítulo anterior. Quando tudo estiver pronto, você deverá trocar a chamada direta à função `iniciar` pelo evento `deviceready` do Cordova:

```

var canvas, context;

window.onload = function() {
    // Troque as linhas quando for fazer deploy no aparelho
    //document.addEventListener('deviceready', iniciar);
    iniciar();
}

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

    // Inicializações virão aqui

    requestAnimationFrame(animar);
}

```

A função de animação, primeiramente, cuidará de limpar a tela e chamar o próximo ciclo:

```

function animar() {
    context.clearRect(0, 0, canvas.width, canvas.height);
}

```

```
// desenhos virão aqui  
requestAnimationFrame(animar);  
}
```

Antes de continuarmos, abra a página no browser e verifique se não aparecem erros no Console — só para garantir que tudo foi digitado corretamente: nomes de variáveis, parênteses etc. O browser dará falta do arquivo `cordova.js`, mas este erro pode ser ignorado até o momento em que for necessário usar alguma API própria do Cordova.

Posicionando e desenhando

Vamos definir agora a posição do objeto. Inicialmente, desenharemos um círculo simples para testar os eventos de touch e, em seguida, o substituiremos pela bruxa.

Declare as variáveis `x` , `y` e `raio` junto com as outras globais da página:

```
var canvas, context;  
var x, y, raio = 32;
```

Na função `iniciar` , comece posicionando o objeto no centro do Canvas e configurando os eventos `touchstart` , `touchmove` e `touchend` :

```
function iniciar() {  
    canvas = document.getElementById('touch');  
    context = canvas.getContext('2d');  
  
    x = canvas.width / 2;  
    y = canvas.height / 2;  
  
    canvas.addEventListener('touchstart', touchStart);  
    canvas.addEventListener('touchmove', touchMove);  
    canvas.addEventListener('touchend', touchEnd);
```

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

Desenhe o círculo em animar :

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

    context.save();

    context.fillStyle = 'red';
    context.strokeStyle = 'blue';
    context.lineWidth = 2;

    context.beginPath();
    context.arc(x, y, raio, 0, Math.PI*2);
    context.fill();
    context.stroke();

    context.restore();

    requestAnimationFrame(animar);
}
```

Programação dos eventos

Vamos agora capturar a posição onde o toque iniciou, na função touchStart . O parâmetro de evento recebido possui o atributo changedTouches , que é um array representando todos os dedos atualmente na tela. Os atributos pageX e pageY nos dão a posição do toque *na página*, não relativamente ao Canvas:

```
function touchStart(e) {
    console.log('x: ' + e.changedTouches[0].pageX +
               ', y: ' + e.changedTouches[0].pageY);
}

function touchMove(e) {
}
```

```
function touchEnd(e) {  
}
```

Neste ponto, faça outro teste no navegador: você deverá ver uma bola vermelha, de borda azul (use as cores que desejar).

Os atributos de posição do `changedTouches` são:

- `pageX` e `pageY` : posição relativa à dimensão completa da página (`viewport`), considerando a posição de rolagem;
- `clientX` e `clientY` : posição relativa à área de renderização do browser (sem considerar a rolagem);
- `screenX` e `screenY` : posição relativa à tela do computador ou dispositivo.

Como você deve ter percebido, os eventos touch não nos dão a posição exata de toque *na área do Canvas*, que é o que precisamos. Mas todo elemento de uma página HTML possui os atributos `offsetLeft` e `offsetTop`, que nos dão a posição atual de um elemento dentro do `viewport`. Se nosso Canvas se encontra na posição (10, 10) da página, por exemplo, o toque em (10, 10) na tela deve corresponder a (0, 0) no Canvas. Ou seja, basta **subtrair os offsets**:

Página

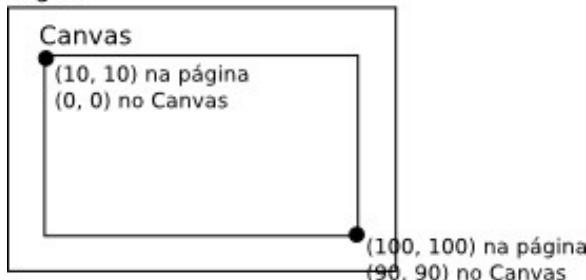


Figura 3.2: Subtraindo os offsets e obtendo a posição de toque dentro de um elemento

Dessa forma, podemos definir as coordenadas do objeto:

```
function touchStart(e) {  
    x = e.changedTouches[0].pageX - canvas.offsetLeft;  
    y = e.changedTouches[0].pageY - canvas.offsetTop;  
    console.log('x: ' + x + ', y: ' + y);  
}
```

Vamos verificar se o toque ocorreu na área do círculo. Para isso, contamos com a ajuda do nosso velho conhecido Teorema de Pitágoras (figura a seguir). A distância em linha reta entre o ponto do toque e a posição atual do círculo corresponde à hipotenusa (a). As distâncias horizontal e vertical correspondem aos catetos (b e c):

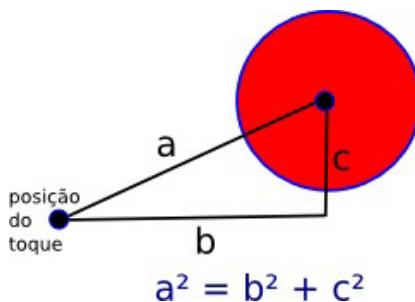


Figura 3.3: Triângulo retângulo formado com a distância entre o ponto tocado e o círculo

Se a hipotenusa for menor ou igual ao raio, tocamos dentro do círculo!

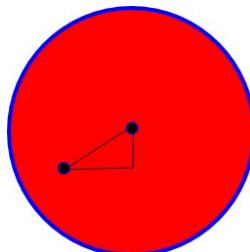


Figura 3.4: Posição de toque dentro da área do círculo: a hipotenusa é menor que o raio

Vamos ao código. Usamos `Math.abs` para obter as distâncias em valores absolutos, pois não importa qual posição é anterior ou posterior a qual (`x - xToque` pode ser negativo se `x < xToque`). Se o toque foi na área do círculo, sinalizamos que o movimento começou (variável `arrastando`), e mudamos as variáveis `x` e `y`:

```
function touchStart(e) {
    var xToque = e.changedTouches[0].pageX - canvas.offsetLeft;
    var yToque = e.changedTouches[0].pageY - canvas.offsetTop;

    var distanciaX = Math.abs(x - xToque);
    var distanciaY = Math.abs(y - yToque);

    // Pitágoras
    if (
        distanciaX*distanciaX + distanciaY*distanciaY <= raio*raio) {
        arrastando = true;
        x = xToque;
        y = yToque;
    }
}
```

Temos de declarar a variável `arrastando` como global:

```
var canvas, context;
```

```
var x, y, raio = 32;  
var arrastando;
```

Para finalizar, na função `touchMove`, verificamos se estamos arrastando a bolinha e atualizamos a posição, igual feito anteriormente. Em `touchEnd`, mudamos a variável `arrastando` para `false`, indicando que a bola não deve mais se mover:

```
function touchMove(e) {  
    if (arrastando) {  
        x = e.changedTouches[0].pageX - canvas.offsetLeft;  
        y = e.changedTouches[0].pageY - canvas.offsetTop;  
    }  
}  
  
function touchEnd(e) {  
    arrastando = false;  
}
```

Vamos testar no navegador. O Google Chrome possui uma forma de emular dispositivos móveis em suas dimensões e eventos touch. Siga os passos:

- Tecle `Ctrl + Shift + J` para abrir o Console.
- Com o foco no Console, tecle `Esc`. Será aberto um novo conjunto de guias.
- Na guia `Emulation`, selecione um modelo pré-configurado qualquer e clique em `Emulate`.
- Atualize a página (`F5`).



Figura 3.5: Emulando dispositivos móveis no Google Chrome

Você pode agora provocar eventos de toque com o mouse no Chrome! Infelizmente, no Android ainda não vai funcionar, mas é por causa de um bug. Acrescente as seguintes linhas ao evento `touchstart` :

```
function touchStart(e) {  
    // ...  
  
    if( navigator.userAgent.match(/Android/i) )  
        e.preventDefault();  
}
```

Pode fazer `cordova run android` e puxar a bolinha com o dedo!

3.2 CONTROLANDO A DIREÇÃO

Para usarmos adequadamente a spritesheet da figura *É um pássaro? É um avião? Use sua imaginação!*, precisamos controlar a direção para onde ocorrem os movimentos, se para a direita ou para a esquerda.

O evento `touchmove` é gerado continuamente conforme nos movimentamos. Para definir se vamos para a direita ou para a

esquerda, temos de guardar a posição no último evento, e comparar com a posição do evento atual. Declare as variáveis `xAnterior` e `yAnterior`:

```
var canvas, context;
var x, y, raio = 32;
var arrastando;
var xAnterior, yAnterior;
```

Guarde nelas a posição inicial do movimento:

```
function touchStart(e) {
    // ...

    // Pitágoras
    if (
        distanciaX*distanciaX + distanciaY*distanciaY <= raio*raio) {
        arrastando = true;
        x = xToque;
        y = yToque;
        xAnterior = x; // Guardar a posição
        yAnterior = y;
    }

    // ...
}
```

E, ao mudar a posição, guarde os valores *anteriores*:

```
function touchMove(e) {
    if (arrastando) {
        xAnterior = x; // Guardar a posição
        yAnterior = y;
        x = e.changedTouches[0].pageX - canvas.offsetLeft;
        y = e.changedTouches[0].pageY - canvas.offsetTop;
    }
}
```

Usando a spritesheet

No pacote de download, no projeto da pasta `touch`, está

presente o arquivo bruxa.png , uma spritesheet com duas imagens 96x96 (192x96 no total). Sabendo as posições atual e anterior, a cada disparo do touchmove , podemos desenhar a figura certa da spritesheet.

Vamos mudar a função animar para chamar outra, desenharBruxa , que cuidará do *clipping* (recorte):

```
function animar() {  
    context.clearRect(0, 0, canvas.width, canvas.height);  
    desenharBruxa();  
    requestAnimationFrame(animar);  
}
```

Nessa nova função, consideraremos x e y , definidos nos eventos de touch, como o centro da imagem. Desta forma, descontamos metade da largura e da altura de destino ao posicionar. Cada bruxa na imagem possui 96x96 pixels, mas vamos renderizá-la menor, em 64x64. Para definir a posição do clipping, comparamos as variáveis x e xAnterior , e assim sabemos se estamos indo para a direita ou para a esquerda:

```
function desenharBruxa() {  
    var largOrigem = 96, largDestino = 64;  
    var xOrigem = (x >= xAnterior)  
        ? 0 // Direita  
        : 96; // Esquerda  
  
    context.drawImage(imgBruxa,  
                    xOrigem, // x na spritesheet  
                    0, // y na spritesheet  
                    largOrigem,  
                    largOrigem,  
                    x - largDestino/2, // x no canvas  
                    y - largDestino/2, // y no canvas  
                    largDestino,  
                    largDestino);  
}
```

Qualquer trabalho análogo na vertical poderia ser feito comparando y com yAnterior .

Precisamos carregar a imagem. Altere a inicialização do app para ficar como segue. Para facilitar agora, podemos manter o raio, deixando a área de touch circular (mas podemos aumentá-lo um pouco para dar mais conforto):

```
var canvas, context;
var x, y, raio = 42; // Aumente um pouco o raio
var arrastando;
var xAnterior, yAnterior;
var imgBruxa;

window.onload = function() {
    imgBruxa = new Image();
    imgBruxa.src = 'bruxa.png';

    // Esperar a imagem carregar
    imgBruxa.onload = function() {
        document.addEventListener('deviceready', iniciar);
        //iniciar();
    }
}
```

Rode o app no dispositivo e leve a bruxinha de um lado para o outro:



Figura 3.6: E lá vamos nós!

3.3 TOQUE EM CANVAS RESPONSIVO

Nosso Canvas está com tamanho fixo, podendo faltar espaço para ele nos dispositivos menores, ou sobrar espaço na página, nos maiores. É interessante podermos usar um pouco de CSS para tornar nosso Canvas responsivo, ou seja, adaptado à tela do dispositivo. Também, como estamos criando apps que rodam localmente, não é nada mau termos imagens um pouco maiores, sendo reduzidas conforme necessário (já que não será feito download delas).

Mas temos um pequeno problema. **A imagem do Canvas sempre é processada com as dimensões informadas na tag.** Em um Canvas responsivo, ela aparece redimensionada. Ao tocar ou clicar, os eventos reportam a posição relativa ao tamanho renderizado em tela, mas na imagem real essas posições sofrem deslocamento, exigindo a conversão entre tamanho da renderização e tamanho da imagem real.

Por mais complicado que pareça, resolver isso é muito simples. Basta fazer uma regra de três. Por exemplo, se o Canvas está sendo renderizado com 960 pixels de largura e eu toco na posição 450 (em x), em qual posição da imagem real eu toquei, se a tag indica 800?

Tamanho	Posição
960	==> 450 (renderizado)
800	==> x (real)

$$x = 800 * 450 / 960$$

O mesmo raciocílio se aplica à posição y .

É fácil descobrir o tamanho com que um elemento da página

HTML é renderizado em tela: basta ler os atributos `offsetWidth` e `offsetHeight`. Assim, podemos ter uma função que recebe as coordenadas capturadas pelos eventos e as transformam em coordenadas da imagem real, através do cálculo descrito:

```
function converterParaCanvas(xToque, yToque) {  
    return {  
        x: canvas.width * xToque / canvas.offsetWidth,  
        y: canvas.height * yToque / canvas.offsetHeight  
    }  
}
```

Devemos também alterar as funções que capturam eventos para chamar a função de conversão. Modifique `touchStart`:

```
function touchStart(e) {  
    var posicao = converterParaCanvas(  
        e.changedTouches[0].pageX - canvas.offsetLeft,  
        e.changedTouches[0].pageY - canvas.offsetTop  
    );  
    var xToque = posicao.x;  
    var yToque = posicao.y;  
  
    // O restante continua igual  
}
```

E `touchMove`:

```
function touchMove(e) {  
    if (arrastando) {  
        xAnterior = x; // Guardar a posição  
        yAnterior = y;  
  
        var posicao = converterParaCanvas(  
            e.changedTouches[0].pageX - canvas.offsetLeft,  
            e.changedTouches[0].pageY - canvas.offsetTop  
        );  
        x = posicao.x;  
        y = posicao.y;  
    }  
}
```

Dimensionando o Canvas

Agora podemos dimensionar o Canvas de forma que fique responsivo, sem causar problemas. Para isso, coloque uma tag `<style>` na seção `<head>` (ou crie um link para um arquivo `.css`).

Na folha de estilo, defina o corpo da página (`body`) como ocupando toda a área disponível do browser (ou webview, em nosso caso), pois ela servirá como uma referência para o Canvas se dimensionar:

```
body {  
    margin: 0;  
    width: 100%;  
    height: 100%;  
}
```

Vamos dimensionar o Canvas, primeiramente, tendo em mente a posição retrato (aparelho em pé), ou seja, com a largura menor que a altura. Ocupamos toda a largura (`width: 100%`;) e mantemos a altura na proporção para a imagem não ficar distorcida (`height: auto;`):

```
/* Padrão (largura < altura) */  
canvas {  
    width: 100%;  
    height: auto;  
    display: block;  
}
```

Para a posição paisagem (deitado), faremos uma *media query* (consulta do dispositivo) que determina uma razão mínima entre a largura e a altura. Acima dessa razão, valerá a nova formatação. Esta razão deve ser passada na forma de fração, por exemplo:

- 2/1 indica que a largura é o dobro da altura;

- 1/2 indica o contrário, que a largura é a metade da altura;
- 1/1 indica uma área quadrada (ambos os lados iguais).

Qualquer fração pode ser usada (como 322/423), mas proporções simples são mais fáceis de mentalizar. Tudo que for abaixo de 1/1 é retrato, e acima é paisagem. Como nossa formatação padrão trata o layout em retrato, vamos estabelecer o mínimo de 1/1 para a formatação de paisagem, por meio do atributo `min-aspect-ratio` da media query:

```
/* A partir de 1/1 (iguais ou largura > altura) */
@media (min-aspect-ratio: 1/1) {  
}
```

Dentro da media query, basta fazer o contrário da formatação padrão: ocupar 100% da altura e ajustar a largura proporcionalmente. Também podemos dar uma margem automática para centralizar o Canvas:

```
/* A partir de 1 (iguais ou largura > altura) */
@media (min-aspect-ratio: 1/1) {
    canvas {
        height: 100%;
        width: auto;
        margin: 0 auto;
    }
}
```

Também podemos adicionar `background-color` ou `background-image` para compor o cenário, fica a seu critério. Vamos brincar um pouco!



Figura 3.7: Canvas se ajustando ao tamanho do dispositivo

WEBDESIGN RESPONSIVO E MOBILE

Embora seja importante na criação de jogos, não é o objetivo aqui ficar falando sobre webdesign responsivo. Caso queira saber em detalhes, você pode ler o livro de Tárcio Zemel (2013):

<http://www.casadocodigo.com.br/products/livro-web-design-responsivo>

Também recomendo muito a leitura do *A Web Mobile*, de Sérgio Lopes (2013):

<http://www.casadocodigo.com.br/products/livro-web-mobile>

Ambos foram fontes de referência fundamentais para este livro.

CAPÍTULO 4

INCREMENTANDO O JOGO DE NAVE

Este capítulo é dedicado a transformar o jogo construído no meu primeiro livro em aplicação móvel. Você aprenderá a criar um controle direcional suave para touchscreen e conhecerá as APIs do Cordova para multimídia e acelerômetro.

Caso você não o conheça, pode jogá-lo agora, em seu desktop ou notebook, em: <http://edyknopfler.github.io/games-js/index.html>. É um jogo de nave e tiro, em que sua missão é destruir discos voadores e não se deixar ser atingido.

Os arquivos-fonte desse jogo podem ser obtidos em:

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

Dentro do `zip`, as pastas estão numeradas pelos capítulos. Procure pegar os arquivos do último capítulo (pasta `09`), pois estão em suas versões finais.

4.1 REVISÃO DAS CLASSES DO JOGO

O jogo de nave possui um pequeno game engine composto pelas seguintes classes de utilidade geral:

- **Animacao** : o coração do game engine. Realiza o game loop e interage com os sprites e processamentos gerais associados a ele.
- **Teclado** : realiza a captura de comandos pelo teclado. Não será usado aqui; em seu lugar criaremos a classe **Direcional** .
- **Colisor** : detecta a colisão entre os sprites associados a ele.
- **Spritesheet** : anima folhas de sprites, dividindo a imagem em quadros de igual tamanho e selecionando o quadro atual de acordo com o tempo e a velocidade da animação.

Já as seguintes classes eram específicas do jogo de nave, mas com o objetivo de demonstrar como os sprites trabalham em um jogo. Tinham de possuir os métodos `atualizar` e `desenhar` , para que a `Animacao` pudesse interagir com eles:

- **Fundo** : faz a rolagem das imagens do cenário.
- **Nave** : a nave controlada pelo jogador. Realizava constantemente a leitura do `Teclado` para se movimentar.
- **Ovni** : o inimigo a ser destruído. Retirava-se da animação após percorrer toda a tela, para não se acumular na memória.
- **Tiro** : disparo da nave. Também se retirava da animação ao sair da tela.
- **Painel** : mostrador de vidas e pontuação.

A maioria destas classes manterá seu funcionamento idêntico na versão mobile do jogo.

4.2 ESQUELETO DO PROJETO

Vamos começar criando o projeto e adicionando a plataforma Android:

```
cordova create nave-mobile-1  
cd nave-mobile-1  
cordova platform add android
```

Depois, copie para a pasta `www` os scripts que você baixou (com exceção de `teclado.js`), e também as pastas `img` e `snd` (imagens e sons).

Façamos uma configuração para travar o jogo na posição retrato (em pé) do aparelho. Por padrão, por ser página da web, um app Cordova acompanha a orientação da tela conforme viramos o dispositivo. Para fixar a tela em uma determinada orientação, podemos inserir uma tag `<preference>` no arquivo `config.xml` . A tag aceita os valores `portrait` (em pé), `landscape` (deitado) e `default` (o mesmo que não usá-la, permitindo girar livremente a tela):

```
<?xml version='1.0' encoding='utf-8'?>  
<widget ...>  
  
...  
  
<preference name="Orientation" value="portrait" />  
</widget>
```

O arquivo `index.html` do projeto vai linkar os scripts `cordova.js` , `rAF.js` (ou como você nomeou o polyfill para `requestAnimationFrame`) e os scripts que você baixou. No corpo, temos o Canvas e o link para o botão `Jogar` :

```
<!DOCTYPE html>  
<html>
```

```

<head>
    <meta charset="utf-8">
    <title>Jogo de Nave - controlado via touch</title>
    <meta name="viewport" content="width=device-width,
        height=device-height, user-scalable=no,
        initial-scale=1, maximum-scale=1, minimum-scale=1">

    <script src="cordova.js"></script>
    <script src="rAF.js"></script>
    <script src="animacao.js"></script>
    <script src="spritesheet.js"></script>
    <script src="colisor.js"></script>
    <script src="fundo.js"></script>
    <script src="nave.js"></script>
    <script src="ovni.js"></script>
    <script src="painel.js"></script>
    <script src="tiro.js"></script>
    <script src="explosao.js"></script>

    <script>
        // A aventura vai começar no celular!
    </script>
</head>

<body>
    <canvas id="jogo_nave" width="500" height="500"></canvas>

    <div id="link_jogar">
        <a href="javascript: iniciarJogo()">Jogar</a>
    </div>
</body>

</html>

```

As variáveis da página são praticamente as mesmas já presentes no jogo. A inicialização é que mudará um pouco, para ficar no formato do Cordova:

```

var canvas, context;
var imagens, animacao, colisor, nave, painel, criadorInimigos;
var totalImagens = 0, carregadas = 0;

```

```

window.onload = function() {
    //document.addEventListener('deviceready', iniciar);
    iniciar();
}

function iniciar() {
    canvas = document.getElementById('jogo_nave');
    context = canvas.getContext('2d');
    carregarImagens();
}

```

A partir daqui, vamos pegar as funções presentes no jogo original, uma por uma. A página HTML está no arquivo `jogo-definitivo.html`, na pasta do capítulo 09. Em alguns casos, vou indicar modificações. Para facilitar, você pode copiar e colar os códigos para o projeto Cordova, e depois adaptá-los. A função de carregamento das imagens permanece a mesma:

```

function carregarImagens() {
    imagens = {
        espaco: 'fundo-espaco.png',
        estrelas: 'fundo-estrelas.png',
        nuvens: 'fundo-nuvens.png',
        nave: 'nave-spritesheet.png',
        ovni: 'ovni.png',
        explosao: 'explosao.png'
    };

    for (var i in imagens) {
        var img = new Image();
        img.src = 'img/' + imagens[i];
        img.onload = carregando;
        totalImagens++;
        imagens[i] = img;
    }
}

```

Para podermos ir mais direto ao que nos interessa agora, que é a transformação do jogo em app mobile, simplifiquei algumas funções. Uma delas é a função que verifica quais imagens já

carregaram, pois aqui o carregamento é mais rápido. Sinta-se livre para recriar a barra de progresso (os códigos estão lá!) ou quaisquer efeitos visuais que desejar: *splashscreens*, *logos*. Estando todas as imagens carregadas, o botão `Jogar` deverá surgir:

```
function carregando() {
    carregadas++;

    if (carregadas == totalImagens) {
        iniciarObjetos();
        document.getElementById('link_jogar').style.display =
            'block';
    }
}
```

A inicialização dos objetos do jogo também mudou pouco. Não há mais o objeto `teclado`, e por isso também a `nave` não o recebe mais no construtor:

```
function iniciarObjetos() {
    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);

    // O construtor da nave não recebe mais o teclado
    nave = new Nave(context, imagens.nave, imagens.exploracao);

    painel = new Painel(context, nave);

    animacao.novoSprite(espaco);
    animacao.novoSprite(estrelas);
    animacao.novoSprite(nuvens);
    animacao.novoSprite(painel);
    animacao.novoSprite(nave);

    colisor.novoSprite(nave);
    animacao.novoProcessamento(colisor);

    configuracoesIniciais();
```

```
}
```

Mudando o construtor da Nave (em nave.js):

```
function Nave(context, imagem, imgExplosao) {
    this.context = context;
    //this.teclado = teclado;

    // ...
}
```

Vamos, por enquanto, comentar os trechos em que a Nave faz uso do teclado. No método `atualizar`:

```
atualizar: function() {
    var incremento =
        this.velocidade * this.animacao.decorrido / 1000;

    /*
    if (this.teclado.pressionada(SETA_ESQUERDA) && this.x > 0)
        this.x -= incremento;

    if (this.teclado.pressionada(SETA_DIREITA) &&
        this.x < this.context.canvas.width - 36)
        this.x += incremento;

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

    if (this.teclado.pressionada(SETA_ABAIXO) &&
        this.y < this.context.canvas.height - 48)
        this.y += incremento;
    */
},
```

E também em `desenhar`:

```
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();
},

```

De volta ao `index.html`, as configurações iniciais também não mudam:

```

function configuracoesIniciais() {
    espaco.velocidade = 60;
    estrelas.velocidade = 150;
    nuvens.velocidade = 500;

    nave.posicionar();
    nave.velocidade = 200;

    criacaoInimigos();

    nave.acabaramVidas = function() {
        animacao.desligar();
        gameOver();
    }

    colisor.aoColidir = function(o1, o2) {
        if ( (o1 instanceof Tiro && o2 instanceof Ovni) ||
            (o1 instanceof Ovni && o2 instanceof Tiro) )
            painel.pontuacao += 10;
    }
}

```

E nem a inicialização do objeto que cria inimigos a cada segundo:

```

function criacaoInimigos() {
    criadorInimigos = {
        ultimoOvni: Date.now(),

        processar: function() {
            var agora = Date.now();
            var decorrido = agora - this.ultimoOvni;

```

```

        if (decorrido > 1000) {
            novoOvni();
            this.ultimoOvni = agora;
        }
    };
};

animacao.novoProcessamento(criadorInimigos);
}

```

Este chama a criação do disco voador, que usava `Math.random` para sortear uma velocidade e uma posição, lembra-se?

```

function novoOvni() {
    var imgOvni = imagens.ovni;
    var ovni = new Ovni(context, imgOvni, imagens.explosao);

    ovni.velocidade =
        Math.floor( 500 + Math.random() * (1000 - 500 + 1) );

    ovni.x =
        Math.floor(Math.random() *
            (canvas.width - imgOvni.width + 1) );

    ovni.y = -imgOvni.height;

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

```

Até aqui, faça um pequeno teste no navegador: ao carregar a página, o único erro no Console deverá ser a falta do `cordova.js`. Mas ainda não dá para clicar em `Jogar`, pois não há a função `iniciarJogo`. Também resolvi simplificá-la; a música será programada posteriormente pela API do Cordova:

```

function iniciarJogo() {
    criadorInimigos.ultimoOvni = Date.now();

    document.getElementById('link_jogar').style.display =

```

```

    'none';

painel.pontuacao = 0;
animacao.ligar();
}

```

Da mesma forma, também simplifiquei a função que finaliza o jogo:

```

function gameOver() {
    document.getElementById('link_jogar').style.display ='block';

nave.vidasExtras = 3;
nave.posicionar();

// A nave é reintroduzida por ter sido retirada na explosão
animacao.novoSprite(nave);
colisor.novoSprite(nave);

removerInimigos();

alert('Game Over');
}

```

Ela chama uma outra função para remover elementos que sobraram na tela:

```

function removerInimigos() {
    for (var i in animacao.sprites) {
        if (animacao.sprites[i] instanceof Ovni ||
            animacao.sprites[i] instanceof Tiro)
            animacao.excluirSprite(animacao.sprites[i]);
    }
}

```

Vamos ao CSS. O Canvas ocupará toda a largura, deixando o restante da altura para o direcional. O botão Jogar inicia-se oculto, e o link dentro dele terá o formato da imagem botao-jogar.png , que já deve estar presente na subpasta www/img :

```

body {
    margin: 0;

```

```
width: 100%;  
height: 100%;  
}  
  
#jogo_nave {  
    width: 100%;  
    height: auto;  
}  
  
#link_jogar {  
    display: none;  
    text-align: center;  
    padding: 20px;  
}  
  
#link_jogar a {  
    color: yellow;  
    background: url(img/botao-jogar.png);  
    font-size: 20px;  
    font-family: sans-serif;  
    text-decoration: none;  
    text-shadow: 2px 2px 5px black;  
  
    display: inline-block;  
    width: 52px;  
    height: 26px;  
    padding: 23px 10px;  
}
```

Faça outro teste no browser. Pode ser preciso rolar a página para encontrar o botão `Jogar`. A ação já deverá acontecer do começo ao fim, do botão `Jogar` ao `Game Over`, embora ainda não se consiga controlar a nave. Mas não é possível rodar no celular, pois a webview padrão do Android ainda não suporta a API de áudio do HTML5 (<http://caniuse.com/audio-api>). Vamos usar a API do Cordova para garantir maior compatibilidade, embora ela não esteja dentro da especificação da W3C.

4.3 API DE SOM DO CORDOVA

Como vimos, o Cordova possui uma API própria para tratar arquivos de multimídia. A partir do Cordova 3.0, as APIs foram quebradas em plugins, devendo cada um ser instalado no projeto para poder ser usado.

APIs do CORDOVA

Você pode encontrar a lista oficial de plugins em:
<http://goo.gl/aP905E>

Encurtei a URL por ser um endereço bastante longo.

Para instalar o plugin de multimídia, abra o terminal/prompt, navegue até a pasta principal do projeto e dê o comando a seguir:

```
cordova plugin add org.apache.cordova.media
```

Esse plugin depende de outro, de acesso a arquivos, e a saída do comando no console indicará que ambos foram instalados:

```
Fetching plugin "org.apache.cordova.media" via plugin registry
Installing "org.apache.cordova.media" for android
Fetching plugin "org.apache.cordova.file" via plugin registry
Installing "org.apache.cordova.file" for android
```

Os sons do jogo estão na pasta sons do projeto no pacote de download. Mova-os para a subpasta platforms/android/assets do projeto Cordova (a partir da pasta principal). Arquivos que serão acessados programaticamente através de código nativo Android podem ficar nessa pasta.

No arquivo `tiro.js`, as linhas iniciais do script carregam o som do disparo. Remova-as:

```
// Remova estas linhas
var SOM_TIRO = new Audio();
SOM_TIRO.src = 'snd/tiro.mp3';
SOM_TIRO.volume = 0.2;
SOM_TIRO.load();
```

No construtor, executamos o som cada vez que um tiro é criado. Também remova as linhas indicadas:

```
function Tiro(context, nave) {
    // ...

    // Remova estas também
    SOM_TIRO.currentTime = 0.0;
    SOM_TIRO.play();
}
```

O mesmo deverá ser feito para o arquivo `explosao.js`.

Agora, acrescente as variáveis que referenciarão os sons no `index.html`:

```
var canvas, context;
var imagens, animacao, colisor, nave, criadorInimigos;
var totalImagens = 0, carregadas = 0;
var musicaAcao, somTiro, somExplosao;
```

Na função `iniciar`, chame `carregarSons`:

```
function iniciar() {
    canvas = document.getElementById('jogo_nave');
    context = canvas.getContext('2d');
    carregarImagens();
    carregarSons();
}
```

E crie essa função. Na API de multimídia do Cordova, um objeto `Media` recebe o arquivo de som no construtor. Na plataforma Android, arquivos na pasta `assets` podem ser indicados escrevendo `file:///android_asset/meu_arquivo`:

```

function carregarSons() {
    musicaAcao =
        new Media('file:///android_asset/musica-acao.mp3');
    somTiro = new Media('file:///android_asset/tiro.mp3');
    somExplosao =
        new Media('file:///android_asset/explosao.mp3');
}

```

Na função `iniciarJogo`, inicie a reprodução do som. O método `seekTo` seleciona a posição no som em milissegundos — setamos para zero para voltar a música no início. Depois é só chamar `play`:

```

function iniciarJogo() {
    // ...

    musicaAcao.seekTo(0);
    musicaAcao.play();
}

```

Na função `gameOver`, mande pausar:

```

function gameOver() {
    musicaAcao.pause();

    // ...
}

```

Os eventos de colisão estão em `configuracoesIniciais`. Acrescente a sequência `seekTo - play` nos eventos de colisão:

```

function configuracoesIniciais() {
    // ...

    // Pontuação e som das explosões
    colisor.aoColidir = function(o1, o2) {
        // Tiro com Ovni
        if ( (o1 instanceof Tiro && o2 instanceof Ovni) ||
            (o1 instanceof Ovni && o2 instanceof Tiro) ) {
            painel.pontuacao += 10;
            somExplosao.seekTo(0);
            somExplosao.play();
        }
    }
}

```

```
    }

    // Nave com Ovni
    if ( (o1 instanceof Nave && o2 instanceof Ovni) ||
        (o1 instanceof Ovni && o2 instanceof Nave) ) {
        somExplosao.seekTo(0);
        somExplosao.play();
    }
}
```

Já é possível rodar o projeto no dispositivo móvel, com o comando `cordova run android`.

OBSERVAÇÕES

Sempre que for testar algo no celular, não se esqueça de mudar o script de inicialização para começar no `deviceready`:

```
window.onload = function() {
    document.addEventListener('deviceready', iniciar);
    //iniciar();
}
```

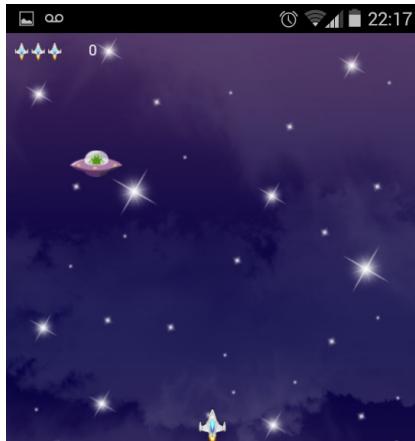


Figura 4.1: Nosso velho jogo de nave virando um app móvel e portável

4.4 CRIANDO UM CONTROLE DIRECIONAL

Para controlar a nave, primeiro vamos criar um controle direcional para telas touch. Ou seja, recriar através de ícones a jogabilidade dos maravilhosos joysticks dos anos 90!



Figura 4.2: Controle do console SNES, da Nintendo

Nesses controles, os quatro botões unidos fisicamente em cruz ou círculo faziam com que, levando o dedo para a diagonal, dois deles ficassem pressionados juntos, forçando a direção diagonal correspondente:



Figura 4.3: Pressionando duas direções juntas para ir para a diagonal

Em nosso caso, como não há união física entre os botões, todas as direções deverão possuir seu ícone correspondente, inteiramente contido em sua célula, sem invadir a área de outro. A célula do centro servirá como posição de descanso do dedo:

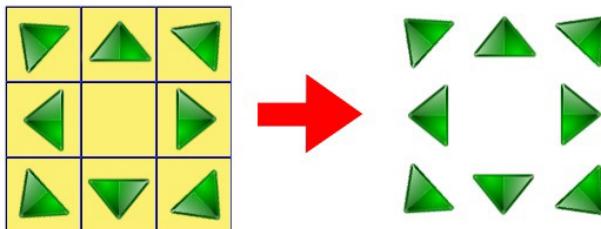


Figura 4.4: Criando um controle direcional

No pacote de downloads *deste* livro (<https://github.com/EdyKnopfler/games-js-2/archive/master.zip>), está a pasta relativa a este projeto, `nave-mobile-1`. Na subpasta `www/img`, se encontra o arquivo `direcional.png`. Copie-o para a `www/img` do seu projeto.

Ao final da tag `<body>`, acrescente uma nova `div` contendo esse direcional e um botão para disparo:

```
<div id="controles">
    
    
</div>
```

Seu posicionamento será feito via CSS. A `div` `controles` inicialmente se encontra oculta, pois só deverá aparecer quando clicarmos no botão `Jogar`. O direcional é posicionado no canto inferior esquerdo e o botão, no inferior direito, um pouco afastado das bordas da tela:

```
#controles {
    display: none;
}

#direcional {
    position: absolute;
```

```
    bottom: 0;
    left: 0;
}

#disparo {
    position: absolute;
    right: 40px;
    bottom: 64px;
}
```

Sinta-se livre para mudar as posições e tamanhos, sem se esquecer que as medidas reais das células influirão na programação posteriormente.

Quando o jogo se inicia, os controles deverão aparecer. Acrescente o comando na função `iniciarJogo` :

```
document.getElementById('controles').style.display =
    'block';
```

E também os faça desaparecer em `gameOver` :

```
document.getElementById('controles').style.display =
    'none';
```

Configurar o disparo é fácil, basta capturar o evento `touchstart`. Em `configuracoesIniciais`, faça:

```
function configuracoesIniciais() {
    // ...

    // Disparo
    document.getElementById('disparo').addEventListener(
        'touchstart',
        function() {
            nave.atirar();
            somTiro.seekTo(0);
            somTiro.play();
        }
    );
}
```

Já dá para atirar e destruir uns OVNIs! Mas vamos iniciar o direcional em `iniciarObjetos`. Seu construtor receberá a imagem na página. Ele também será passado para o construtor da Nave, como era feito com o Teclado na versão desktop do jogo:

```
function iniciarObjetos() {  
    // ...  
    direcional = new Direcional(  
        document.getElementById('direcional'));  
    nave = new Nave(context, direcional, imagens.nave,  
        imagens.exploracao);  
    // ...  
}
```

Criemos agora a classe `Direcional` no arquivo `direcional.js`. Primeiro, definimos nomes para as direções e recebemos a imagem a ser usada no construtor. Os valores não importam, contanto que sejam únicos:

```
var DIRECAO_NENHUMA = 0;  
var DIRECAO_N = 1;      // norte  
var DIRECAO_S = 2;      // sul  
var DIRECAO_L = 3;      // leste  
var DIRECAO_O = 4;      // oeste  
var DIRECAO_NE = 5;     // nordeste  
var DIRECAO_NO = 6;     // noroeste  
var DIRECAO_SE = 7;     // sudeste  
var DIRECAO_SO = 8;     // sudoeste  
  
function Direcional(imagem) {  
    this.imagem = imagem;  
    this.direcao = DIRECAO_NENHUMA;  
  
    // continua...  
}
```

O que temos de fazer é atribuir eventos de toque à imagem. Vamos atribuir as funções de evento a variáveis, de forma que cada uma possa ser usada em mais de um evento. Uma delas é

responsável por capturar as coordenadas, já descontando a posição da imagem (*offsets*), passando-as para que o método `direcaoPonto`, a ser criado logo mais, determine a direção. A outra servirá para finalizar o movimento, atribuindo `DIRECAO_NENHUMA` ao direcional:

```
function Direcional(imagem) {
    // ...

    var direcional = this;
    var funcaoToque = function(e) {
        direcional.direcaoPonto(
            e.changedTouches[0].pageX - imagem.offsetLeft,
            e.changedTouches[0].pageY - imagem.offsetTop
        );
    }
    var funcaoCancela = function() {
        direcional.direcao = DIRECAO_NENHUMA;
    }

    // continua...
}
```

Tendo as funções em mãos, vamos atribuí-las a todos os eventos descritos no início do capítulo *Eventos touch*. Na seção *Arrastando um objeto* (capítulo anterior) foi dito que, no Android, temos de chamar o `preventDefault` no `touchstart` para contornar um bug que impede o `touchmove` de ocorrer:

```
function Direcional(imagem) {
    // ...

    imagem.addEventListener('touchstart', function(e) {
        funcaoToque(e);
        if( navigator.userAgent.match(/Android/i) )
            e.preventDefault();
    });
    imagem.addEventListener('touchmove', funcaoToque);
    imagem.addEventListener('touchend', funcaoCancela);
    imagem.addEventListener('touchleave', funcaoCancela);
```

```
    imagem.addEventListener('touchcenter', funcaoToque);
    imagem.addEventListener('touchcancel', funcaoCancela);
}
```

Permitir continuar mudando a direção no touchmove tornará nosso direcional mais suave, sem exigir que o jogador tire o dedo de um ícone para tocar outro. Já tentei jogar jogos em que não tomaram esse cuidado e achei a experiência muito frustrante.

A função que recebe o comando de toque passa a responsabilidade de determinar a direção para o método direcaoPonto . Este divide a área da imagem em três partes iguais, tanto na largura quanto na altura, e determina qual célula foi tocada. Veja como foi testada a primeira coluna e, dentro dela, testamos pela altura as direções noroeste, oeste e sudoeste:

```
Direcional.prototype = {
  direcaoPonto: function(x, y) {
    var larguraCelula = this.imagem.width / 3;
    var alturaCelula = this.imagem.height / 3;

    // 1ª coluna
    if (x < larguraCelula) {
      // Verifica qual linha
      if (y < alturaCelula)
        this.direcao = DIRECAO_NO;
      else if (y < alturaCelula * 2)
        this.direcao = DIRECAO_O;
      else
        this.direcao = DIRECAO_SO;
    }

    // continua...
  }
}
```

Seguindo este raciocínio, na segunda coluna, testamos o norte, a posição de descanso e o sul; e na terceira testamos o nordeste, o leste e o sudoeste:

```

// 2ª coluna
else if (x < larguraCelula * 2) {
    if (y < alturaCelula)
        this.direcao = DIRECAO_N;
    else if (y < alturaCelula * 2)
        this.direcao = DIRECAO_NENHUMA; // Centro do direcional
    else
        this.direcao = DIRECAO_S;
}

// 3ª coluna
else {
    if (y < alturaCelula)
        this.direcao = DIRECAO_NE;
    else if (y < alturaCelula * 2)
        this.direcao = DIRECAO_L;
    else
        this.direcao = DIRECAO_SE;
}

```

A classe `Direcional` está pronta, temos agora de incluí-la no jogo. Adicione o script no `<head>`:

```
<script src="direcional.js"></script>
```

Como estamos passando o objeto direcional no construtor da `Nave`, receba-o na `function` da classe e crie o atributo correspondente:

```

function Nave(context, direcional, imagem, imgExplosao) {
    this.context = context;
    this.direcional = direcional;

    // ...
}

```

Da mesma forma que fazia com o `Teclado`, a nave deverá ler o estado do `Direcional` para determinar seus deslocamentos em x e y . Poderíamos testar direção por direção, mas isso daria **muito** trabalho para testarmos também os limites do Canvas em cada caso:

```

// Exemplo teórico
if (dir.direcao == DIRECAO_NO) {
    if (this.x > 0)
        this.x -= incremento;

    if (this.y > 0)
        this.y -= incremento;
}

// outras direções...

```

Em vez disso, vamos testar por esquerda, direita, acima e abaixo isoladamente. Por exemplo: noroeste, oeste e sudoeste são movimentos para a esquerda; ou sudoeste, sul e sudeste são movimentos para baixo etc. Comece testando essas condições no método `atualizar` da Nave :

```

atualizar: function() {
    var incremento =
        this.velocidade * this.animacao.decorrido / 1000;

    var dir = this.direcional;
    var esquerda = dir.direcao == DIRECAO_NO ||
                   dir.direcao == DIRECAO_O ||
                   dir.direcao == DIRECAO_SO;
    var direita = dir.direcao == DIRECAO_NE ||
                  dir.direcao == DIRECAO_L ||
                  dir.direcao == DIRECAO_SE;
    var acima = dir.direcao == DIRECAO_NO ||
                  dir.direcao == DIRECAO_N ||
                  dir.direcao == DIRECAO_NE;
    var abaixo = dir.direcao == DIRECAO_SO ||
                  dir.direcao == DIRECAO_S ||
                  dir.direcao == DIRECAO_SE;

    // continua ...
},

```

Conseguimos reduzir aos quatro casos originais, para saber se há movimento em `x` e/ou em `y`:

```
atualizar: function() {
```

```

// ...

if (esquerda && this.x > 0)
    this.x -= incremento;

if (direita && this.x < this.context.canvas.width - 36)
    this.x += incremento;

if (acima && this.y > 0)
    this.y -= incremento;

if (abaixo && this.y < this.context.canvas.height - 48)
    this.y += incremento;
},

```

O método desenhar só precisa testar pela esquerda ou direita, para saber com qual linha da spritesheet vai executar a animação:

```

desenhar: function() {
    var dir = this.direcional;

    if (dir.direcao == DIRECAO_NO ||
        dir.direcao == DIRECAO_O ||
        dir.direcao == DIRECAO_SO)
        this.spritesheet.linha = 1;

    else if (dir.direcao == DIRECAO_NE ||
              dir.direcao == DIRECAO_L ||
              dir.direcao == DIRECAO_SE)
        this.spritesheet.linha = 2;

    else
        this.spritesheet.linha = 0;

    this.spritesheet.desenhar(this.x, this.y);
    this.spritesheet.proximoQuadro();
},

```

Por fim, em `gameOver`, faça com que o movimento cesse, para a nave não sair desembestada na próxima partida. Quando temos janelas de alerta, elas interferem no foco da aplicação, fazendo com

que os eventos touch na imagem não sejam ouvidos. Por exemplo, tirar o dedo para dar "OK" no alerta não força um touchend na imagem. Por isso, façamos os ajustes **antes** do alerta:

```
function gameOver() {  
    // ...  
  
    direcional.direcao = DIRECAO_NENHUMA;  
    alert('Game Over');  
}
```

Tente jogar. Tudo funciona **quase** normalmente... Vejamos por quê.

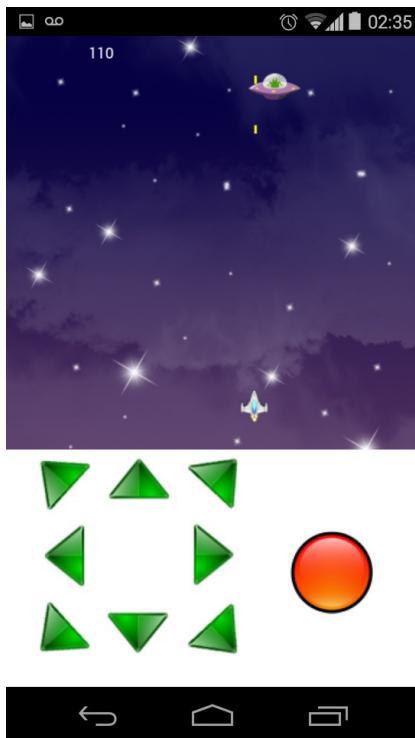


Figura 4.5: Jogo de nave com direcional touch

4.5 PRECISÃO COM MÚLTIPLOS DEDOS

Se você conseguiu jogar, pode acontecer que a nave vá sempre para a direita, não importando em qual posição do direcional você está com o dedo. Não é toda hora que acontece, mas é o suficiente para incomodar e fazer com que alguém dê nota negativa para seu jogo na Play Store. Isto ocorre especificamente se você primeiro der um disparo, mantiver o dedo no botão e, em seguida, tocar o direcional com outro dedo:

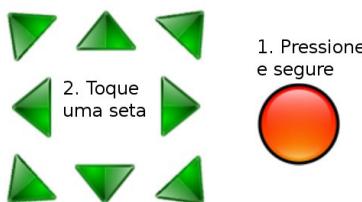


Figura 4.6: Com o dedo no botão de disparo, a nave move-se para a direita!

Lembra-se de que o atributo `changedTouches` do evento é, na verdade, um array com todos os dedos na tela? Se já havia um dedo no botão, ele é o elemento zero; o toque na imagem então cria o elemento número 1. Estamos sempre testando o elemento zero, e como o botão fica mais à direita, nosso algoritmo interpreta que tocamos a terceira coluna:

```
if (x < larguraCelula) {  
    // ...  
}  
else if (x < larguraCelula * 2) {  
    // ...  
}  
else {  
    // ...  
}
```

O primeiro passo é, justamente, colocar um limite final em vez de um `else` simples. No método `direcaoPonto` da classe `Direcional`, modifique o `else` da terceira coluna para incluir a condição:

```
// 3ª coluna
else if (x < larguraCelula * 3) {
    // ...
}
```

Também vale a pena colocarmos limites nos `else`s finais internos, que testam a posição *y*:

```
// 1ª coluna
if (x < larguraCelula) {
    // ...
else if (y < alturaCelula * 3)
    return DIRECAO_SO;
}

// Faça para todas as colunas!
```

Por fim, no construtor, em vez de testar somente a posição zero do `changedTouches`, vamos fazer um loop e testar todo o array. Não importa qual a ordem dos dedos, se algum estiver dentro dos limites de uma célula, deverá ser considerado. **Atenção:** no lugar de zero, usamos a variável de loop *i*:

```
function Direcional(imagem) {
    // ...

    var funcaoToque = function(e) {
        for (var i in e.changedTouches) {
            direcional.direcaoPonto(
                e.changedTouches[i].pageX - imagem.offsetLeft,
                e.changedTouches[i].pageY - imagem.offsetTop
            );
        }
    }
}
```

```
// ...  
}
```

Perceba que, se testarmos os limites exatos de uma área de toque, podemos verificar todo o array de dedos, sem nos preocuparmos com qual elemento do array representa aquele toque. Agora sim nosso controle tem um tratamento mais profissional.

E assim concluímos a primeira versão móvel de nosso jogo de nave! E a aventura só está começando, porque em um dispositivo móvel inúmeras são as possibilidades de jogabilidade.

4.6 API DO ACELERÔMETRO DO CORDOVA

O acelerômetro é o sensor de movimento de seu dispositivo móvel: aquele que detecta se você está usando o aparelho em retrato (em pé) ou paisagem (deitado), por exemplo. Mas ele é capaz de muito mais que isso.

Joysticks como o do Nintendo Wii também são baseados em acelerômetros. O que ele faz é detectar movimentos nos seguintes eixos:

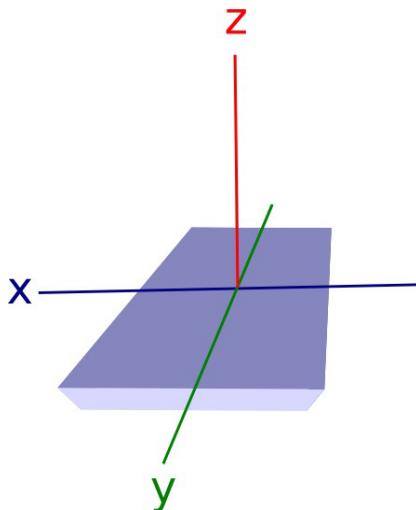


Figura 4.7: Eixos de movimentação monitorados pelo acelerômetro

Podemos copiar toda a pasta `nave-mobile-1` para `nave-mobile-2`, na qual vamos trabalhar agora. Acesse esta pasta do terminal/prompt e adicione o plugin para leitura do acelerômetro:

```
cordova plugin add org.apache.cordova.device-motion
```

Na página `index.html`, deixe somente botão de disparo na `div controles`. Se quiser, você pode deletar deste projeto todos os arquivos relacionados ao direcional (imagem e script), e também a tag `<script src="direcional.js">`.

```
<div id="controles">
  
</div>
```

Na função `iniciarObjetos`, não será mais iniciado o direcional, nem passado para o construtor da Nave :

```
function iniciarObjetos() {
  // ...
```

```
// Remova esta linha  
//direcional = new Direcional(document.getElementById(  
//                                'direcional'));  
  
// Não passe mais o direcional  
nave = new Nave(context, imagens.nave, imagens.explorao);  
  
// ...  
}
```

Remova também, da função `gameOver`, a linha que reseta a direção:

```
direcional.direcao = DIRECAO_NENHUMA;
```

O construtor da classe `Nave` fica um pouco mais enxuto sem ele:

```
function Nave(context, imagem, imgExplorao) {  
    this.context = context;  
  
    // Remova  
    //this.direcional = direcional;  
  
    // ...  
}
```

Deixe, por enquanto, os métodos `atualizar` e `desenhar` com os corpos comentados. Logo mais, esses métodos vão se valer dos dados do acelerômetro. No entanto, eles não farão a sua leitura diretamente, pois fazer isso no loop de animação é **muito** lento. Em vez disso, vamos deixar o código nativo do Cordova monitorar o acelerômetro para nós.

```
atualizar: function() {  
    /* comente */  
},  
desenhar: function() {  
    /* comente */  
},
```

No `index.html`, função `iniciarJogo`, configure logo no início uma chamada a `navigator.accelerometer.watchAcceleration`, que recebe uma função de callback a ser chamada de tempos em tempos. A frequência é configurada em milissegundos, em um objeto no último parâmetro. Fiz vários testes até chegar a um valor que me agradou; você pode experimentar outros intervalos e ajustar conforme cada jogo exigir uma sensibilidade diferente:

```
function iniciarJogo() {
    acelerometro = navigator.accelerometer.watchAcceleration(
        function(aceleracao) {
            var msg =
                'X: ' + aceleracao.x + '<br>' +
                'Y: ' + aceleracao.y + '<br>' +
                'Z: ' + aceleracao.z;

            document.getElementById('controles').innerHTML = msg;
        },
        function() {
            alert('Erro!');
        },
        { frequency: 25 }
    );
}

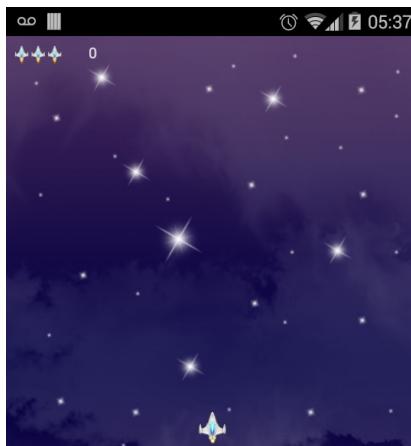
// ...
}
```

A variável `acelerometro` pode ser declarada junto às outras principais do jogo, no começo do script:

```
var canvas, context;
var imagens, animacao, colisor, nave, criadorInimigos;
var totalImagens = 0, carregadas = 0;
var musicaAcao, somTiro, somExplosao;
var acelerometro;
```

Rode o projeto e veja os valores. O botão de disparo some porque estamos sobrescrevendo todo o conteúdo da `div` (mas

isso é temporário):



X: -0.09576806426048279
Y: 1.6855180263519287
Z: 8.925583839416504

Figura 4.8: Obtendo dados do acelerômetro

Balance o aparelho e confira os valores conforme a direção de inclinação:

- **Esquerda:** $x > 0$
- **Direita:** $x < 0$
- **Cima:** $y < 0$
- **Baixo:** $y > 0$

Para nossos propósitos, o eixo z não é necessário. Podemos

então passar os valores obtidos para a nave:

```
function iniciarJogo() {
    acelerometro = navigator.accelerometer.watchAcceleration(
        function(aceleracao) {
            nave.aceleracao.x = aceleracao.x;
            nave.aceleracao.y = aceleracao.y;
        },
        function() {
            alert('Erro!');
        },
        { frequency: 25 }
    );
}

// ...
}
```

Repare que criamos o atributo `aceleracao` na nave. Vamos iniciá-lo no construtor com um objeto contendo os dois eixos usados e o valor zero para cada um:

```
function Nave(context, imagem, imgExplosao) {
    // ...

    this.aceleracao = { x: 0, y: 0 };
}
```

O método `atualizar` fará a leitura dos dados e movimentará a nave conforme o sentido em que balançamos o aparelho, cujos testes vimos há pouco:

```
atualizar: function() {
    var incremento =
        this.velocidade * animacao.decorrido / 1000;
    var acc = this.aceleracao;

    if (acc.x > 0 && this.x > 0)
        this.x -= incremento;

    if (acc.x < 0 && this.x < this.context.canvas.width - 36)
        this.x += incremento;
```

```
    if (acc.y < 0 && this.y > 0)
        this.y -= incremento;

    if (acc.y > 0 && this.y < this.context.canvas.height - 48)
        this.y += incremento;
},
```

O mesmo vale para o método `desenhar` :

```
desenhar: function() {
    var acc = this.aceleracao;

    if (acc.x > 0)
        this.spritesheet.linha = 1;
    else if (acc.x < 0)
        this.spritesheet.linha = 2;
    else
        this.spritesheet.linha = 0;

    this.spritesheet.desenhar(this.x, this.y);
    this.spritesheet.proximoQuadro();
},
```

Na função `gameOver` , vamos parar de monitorar o acelerômetro chamando `navigator.accelerometer.clearWatch` :

```
function gameOver() {
    navigator.accelerometer.clearWatch(accelerometro);

    // ...
}
```

Experimente jogar mexendo o aparelho na mão! É muito divertido, mas uma melhoria ainda pode ser feita.

Calibrando uma posição de repouso

Estamos sempre comparando os valores do acelerômetro com o valor zero. Isto faz com que a posição de referência para o

movimento seja "deitado" na horizontal, o que (pelo menos para mim) provoca algum desconforto por ter de abaixar bem a cabeça para ver a ação. Que tal se, no início do jogo, o jogador puder definir uma posição de repouso como achar mais confortável, e o jogo a usasse como referência para movimentar a nave? Isso é bem fácil de implementar.

Inicialmente, na função `iniciarJogo`, indicamos que queremos fazer a calibragem na primeira leitura de acelerômetro da partida. Quando o jogo começa, setamos uma variável `primeiraLeitura` para `true` e, no callback do acelerômetro, setamos um novo atributo `repouso` na nave com os primeiros valores lidos:

```
function iniciarJogo() {
    primeiraLeitura = true;

    acelerometro = navigator.accelerometer.watchAcceleration(
        function(aceleracao) {
            // Na primeira leitura, calibraremos a nave
            if (primeiraLeitura) {
                nave.repouso.x = aceleracao.x;
                nave.repouso.y = aceleracao.y;
                primeiraLeitura = false;
            }

            nave.aceleracao.x = aceleracao.x;
            nave.aceleracao.y = aceleracao.y;
        },
        function() {
            alert('Erro!');
        },
        { frequency: 25 }
    );
}
```

A variável `primeiraLeitura` deve ser declarada no começo, junto com `acelerometro`:

```
// ...
```

```
var acelerometro, primeiraLeitura;
```

O novo atributo repouso da Nave guardará a posição de referência. Inicialize-o no construtor:

```
function Nave(context, imagem, imgExplosao) {  
    // ...  
    this.repouso = { x: 0, y: 0 };  
}
```

Por último, nos métodos atualizar e desenhar , temos de usar essa posição em vez de zero nas comparações. Onde comparamos acc.x , troque o zero por this.repouso.x . O mesmo para a coordenada y :

```
// Exemplo (troque em todas as comparações)  
  
// Onde era:  
if (acc.x > 0 && this.x > 0)  
    this.x -= incremento;  
  
// Troque por:  
if (acc.x > this.repouso.x && this.x > 0)  
    this.x -= incremento;  
  
// ...
```

Faça o teste e veja se consegue jogar com qualquer inclinação desejada.

4.7 ADAPTANDO O CANVAS AO TAMANHO DA TELA

Em *Toque em Canvas responsivo*, aprendemos a criar um Canvas que se adapta à tela via CSS. Como as imagens geradas eram dimensionadas, isto requeria que cada elemento gráfico tivesse um tamanho razoável, a fim de se adaptar a telas maiores.

Vamos fazer diferente agora: configurar dinamicamente o Canvas para o tamanho exato disponível na tela. Estamos configurando os atributos da tag, `width` e `height`. Os valores vêm dos atributos `availWidth` e `availHeight` do objeto `screen`, que nos dão a largura e a altura da tela, descontadas barras de tarefas, de botões e de ícones indicadores do sistema operacional. Façamos isto na função `iniciar`, onde ocorre a inicialização do app:

```
function iniciar() {
    canvas = document.getElementById('jogo_nave');
    context = canvas.getContext('2d');
    canvas.width = screen.availWidth;
    canvas.height = screen.availHeight;
    carregarImagens();
    carregarSons();
}
```

Como o Canvas vai ocupar a tela toda, precisamos fazer uso do atributo `z-index` do CSS para que os botões `Jogar` e de disparo fiquem por cima. Estes recebem o valor 1, enquanto o Canvas recebe o valor zero:

```
body { /* não mudou */ }

#jogo_nave {
    position: absolute;
    z-index: 0;
}

#link_jogar {
    display: none;
    position: absolute;
    z-index: 1;
    bottom: 20px;
    width: 100%;
    text-align: center;
}
```

```
#link_jogar a { /* não mudou */ }

#controles { /* não mudou */ }

#direcional { /* desaparece */ }

#disparo {
    position: absolute;
    right: 10px;
    bottom: 10px;
    z-index: 1;
}
```

As adaptações ao jogo de nave acabam aqui. Você agora tem uma base bastante robusta para criar jogos móveis em estilo arcade, além de mais diversão para a fila do banco ou do supermercado! No próximo capítulo, daremos início à construção de uma mesa de bilhar móvel, usando o framework `Hammer.js` para controlar as tacadas.

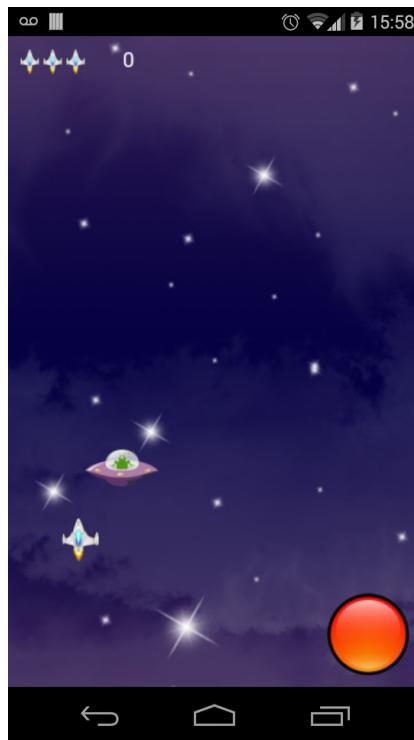


Figura 4.9: Jogo com acelerômetro ocupando a tela

CAPÍTULO 5

INTERAÇÕES AVANÇADAS COM HAMMER.JS

Neste capítulo, você aprenderá a capturar gestos comuns, porém mais complexos, de forma prática com a biblioteca *Hammer.js*. A partir daqui, um pequeno jogo de bilhar começará a tomar forma. Deixaremos o jogador rotacionar o taco com os dedos e regular a força da tacada arrastando um ponteiro na tela.



Figura 5.1: Taco de bilhar touch

5.1 CONHECENDO O HAMMER.JS

Gestos comuns

Você está navegando na internet e surge uma página com letras miúdas. O que você precisa fazer? Dar zoom. Como dar zoom no browser do dispositivo móvel? Com o gesto chamado *pinch* (pinça), que consiste em colocar dois dedos na tela e afastá-los ou aproximá-los. Afastando os dedos, você dá zoom; aproximando, você tira o zoom.

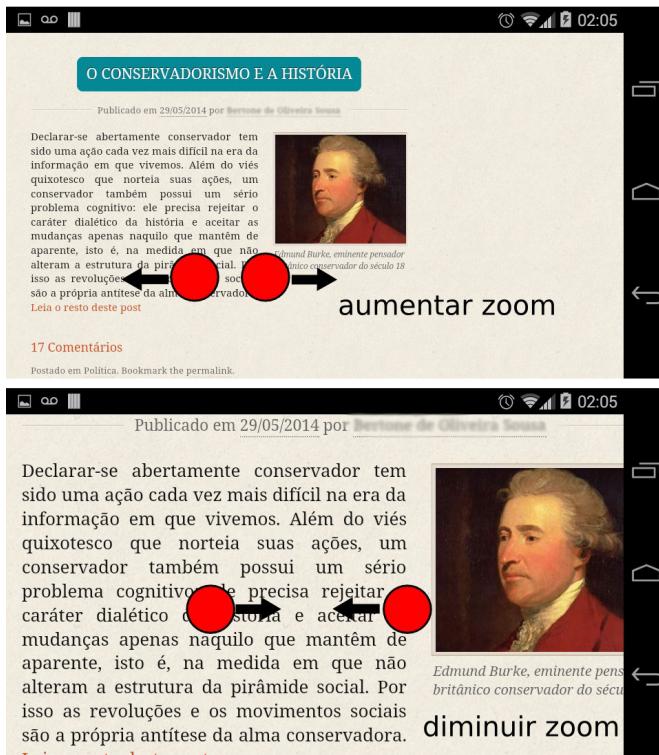


Figura 5.2: Fazendo a pinça para ajustar o zoom

Existe também um gesto usado para girar coisas. O gesto de rotação (*rotate*) é usado, por exemplo, no Google Earth. Você

coloca dois dedos na tela e gira como se estivesse cochando um parafuso:

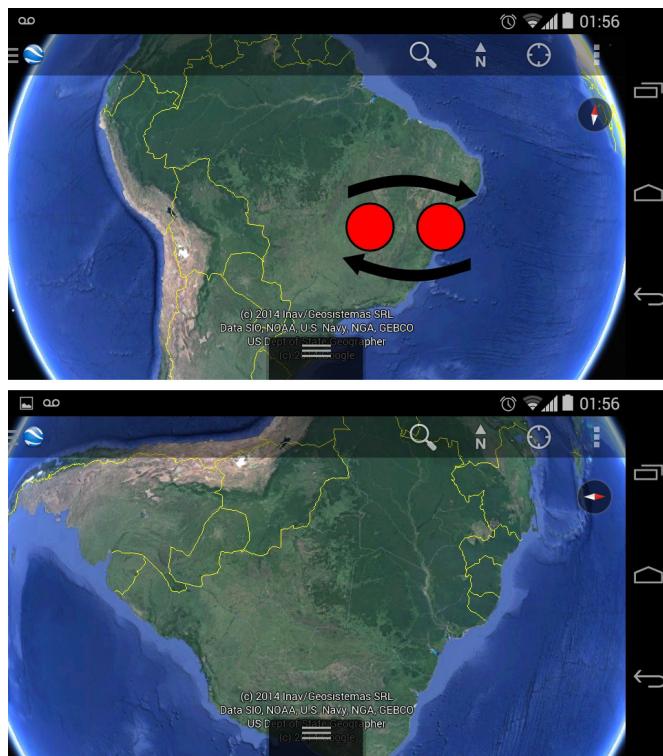


Figura 5.3: Girando o globo com o gesto rotate

Outros gestos usuais são:

- **Tap** (toque): toque rápido;
- **Press** (pressionar): manter o dedo sobre um objeto por um curto instante. Normalmente abre um menu contextual;
- **Swipe** (deslizar): mover o dedo rapidamente de um ponto a outro. Usado para passar fotos na tela, abrir

menus ocultos em apps e virar páginas em e-books;

- **Pan** ou **drag** (arrastar): deslocamento de um objeto com o dedo.

Tecnicamente, a diferença entre o *swipe* e o *pan* é a velocidade: o *swipe* só ocorre a partir de uma velocidade mínima definida.

Implementar esses gestos usando a API de mais baixo nível do HTML5 é relativamente complexo. Para reconhecer um simples *press*, por exemplo, você iniciaria a contagem do tempo no `touchstart` e a interromperia no `touchend`. Um timeout poderia ser disparado se o dedo foi mantido no objeto pelo tempo que você determinar:

```
// Exemplo teórico
elemento.addEventListener('touchstart', function() {
    timeout = setTimeout(press, 500);
});

elemento.addEventListener('touchend', function() {
    clearTimeout(timeout);
});

function press() {
    // Executo minha ação
}
```

Isso porque não estamos considerando se o usuário moveu o dedo ou não. Se quiséssemos evitar, teríamos de cancelar o *timeout* também no `touchmove`. No capítulo *Eventos touch*, implementamos um *pan* para fazer uma bruxa voar pelo Canvas. Se quiséssemos restringir o movimento somente na vertical ou na horizontal, lá se iam mais umas linhas de código. O *pinch* e o *rotate* já são um caso a parte de complexidade.

Felizmente, todos esses eventos já se encontram

implementados na biblioteca *Hammer.js*. E o que é melhor: são reconhecidos não só em telas touch, mas também através do mouse!

Usando o Hammer.js

O download da biblioteca pode ser feito em:
<http://hammerjs.github.io/>.

Tendo linkado o arquivo na página, se quisermos reconhecer o gesto de arrastar, primeiro devemos instanciar um objeto `Hammer`, passando o elemento da página no construtor. Depois, o método `on` manda o `Hammer.js` reconhecer um evento. Passo para ele a string 'pan' e o callback:

```
<script src="hammer.min.js"></script>
<script>
    var canvas = ...

    var hammer = new Hammer(canvas);

    hammer.on('pan', function(e) {
        console.log(e.center.x + ', ' + e.center.y);
    });
</script>
```

Os eventos podem ser desctrinchados: em vez de simplesmente `pan`, poderíamos dizer `panstart`, `panmove` ou `panend`. E melhor ainda: `panleft`, `panright`, `panup`, `pandown`! (`panup` e `pandown` ainda não funcionam, mas logo você saberá por quê.)

EVENTOS RECONHECIDOS PELO HAMMER.JS

Aqui estão os eventos padrão do Hammer.js e seus variantes:

- **tap**
- **press**
- **pan** e *panstart*, *panmove*, *panend*, *pancancel*,
panleft, *panright*, *panup*, *pandown*
- **swipe** e *swipeleft*, *swiperight*, *swipeup*, *swipedown*
- **pinch** e *pinchstart*, *pinchmove*, *pinchend*,
pinchcancel, *pinchin*, *pinchout*
- **rotate** e *rotatestart*, *rotatemove*, *rotateend*,
rotatecancel

Mas calma: nem tudo isso funcionará de imediato. Os eventos **pinch** e **rotate** são desabilitados por padrão, e o **pan** e o **swipe** só detectam inicialmente movimentos na horizontal. O motivo disso é que esses gestos são reservados em páginas web: fazer a pinça para dar zoom e arrastar na vertical para rolar a página. O browser não interpreta o **rotate**, mas os dedos podem se afastar ou aproximar e um **pinch** ocorrerá. No caso de jogos e apps em geral, como estamos sempre restringindo a tela, podemos habilitar esses gestos:

```
hammer.get('pinch').set({ enable: true });
hammer.get('rotate').set({ enable: true });
```

E também habilitar as direções desejadas para o **pan** e o **swipe**:

```
hammer.get('pan').set({ direction: Hammer.DIRECTION_ALL });
hammer.get('swipe')
```

```
.set({ direction: Hammer.DIRECTION_VERTICAL }));
```

As constantes em `Hammer` são:

- DIRECTION_NONE
- DIRECTION_LEFT
- DIRECTION_RIGHT
- DIRECTION_UP
- DIRECTION_DOWN
- DIRECTION_HORIZONTAL
- DIRECTION_VERTICAL
- DIRECTION_ALL

Uma configuração a que devemos prestar atenção é a `threshold`, que define o deslocamento mínimo para um evento ocorrer. No caso do `pan`, o valor padrão é 10 pixels! Podemos setá-la como zero, para um deslocamento suave. Para podermos arrastar um objeto livremente, o código fica:

```
hammer.get('pan').set({
  direction: Hammer.DIRECTION_ALL,
  threshold: 0
});

hammer.on('pan', function(e) {
  // ...
});
```

No callback, um dos atributos mais importantes do evento é `center`, que contém as coordenadas `x` e `y` do ponto central entre os dedos, ou do próprio dedo, se houver um só, como no último exemplo. No caso de um `rotate` ou `pinch`, por exemplo, a posição de cada dedo estaria no atributo `pointers`, que é um array com os dedos na tela:

```
hammer.get('rotate').set({ enable: true });
```

```
hammer.on('rotate', function(e) {
    console.log('Centro: ' + e.center.x + ', ' + e.center.y);

    console.log('Dedo 1: ' + evento.pointers[0].pageX + ', ' +
        evento.pointers[0].pageY);

    console.log('Dedo 2: ' + evento.pointers[1].pageX + ', ' +
        evento.pointers[1].pageY);
});
```

Tendo feito esta introdução ao Hammer.js, podemos começar a utilizá-lo em nosso jogo de bilhar.

DOCUMENTAÇÃO DO HAMMER.JS

Você pode obter mais informações em:
<http://hammerjs.github.io/getting-started/>

Lá se encontra uma descrição detalhada da API e dos eventos (*recognizers*).

5.2 BARRA DE AJUSTE DE FORÇA

Vamos começar pelo mais fácil: uma barra para o jogador ajustar a força da tacada. Ela possuirá um ponteiro arrastável dentro dos seus limites.

Comece criando o projeto `taco-bilhar`, adicione a plataforma Android e adicione o plugin de multimídia:

```
cordova create taco-bilhar
cd taco-bilhar
cordova platform add android
```

```
cordova plugin add org.apache.cordova.media
```

No pacote de download do livro, na pasta `tacobilhar/sons`, existe o arquivo `colisao.mp3`. Este é o som da tacada. Copie-o para a pasta de `platforms/android/assets` do seu projeto. Copie também as imagens da pasta `www/img`.

Também configure no `config.xml` a orientação `landscape` (deitado), e um ícone para o projeto. No download já há um ícone na pasta de imagens. Se desejar, você pode procurar ou criar a imagem que preferir:

```
<preference name="Orientation" value="landscape" />
<icon src="www/img/icone.png" />
```

Coloque na pasta `www` os scripts do `requestAnimationFrame` e do `Hammer.js`. O esqueleto do `index.html` ficará assim:

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,
        height=device-height, user-scalable=no,
        initial-scale=1, maximum-scale=1, minimum-scale=1">
    <title>Taco de Bilhar</title>

    <script src="cordova.js"></script>
    <script src="rAF.js"></script>
    <script src="hammer.min.js"></script>
    <script src="barra-forca.js"></script>
    <script src="taco.js"></script>

    <script>
        // inicialização
    </script>

    <style>
```

```
/* folha de estilo */
</style>
</head>

<body>
<canvas id="taco_bilhar"></canvas>
</body>

</html>
```

Dentro da tag `<style>`, coloque o CSS para ajustar o Canvas conforme aprendemos, deixando-o responsivo de acordo com a altura da tela:

```
body {
  margin: 0;
  width: 100%;
  height: 100%;
  background: #333;
}

canvas {
  height: 100%;
  width: auto;
  display: block;
  margin: 0 auto;
}
```

Fazemos a inicialização do jogo, configurando o Canvas e carregando as imagens e sons. As linhas que não rodam no browser desktop foram comentadas, você poderá descomentá-las depois quando formos testar no celular ou tablet:

```
var canvas, context;
var imgTaco, imgPonteiro, somTacada;
var taco, barraForca;

window.onload = function() {
  canvas = document.getElementById('taco_bilhar');
  context = canvas.getContext('2d');
  canvas.width = screen.availWidth;
  canvas.height = screen.availHeight;
```

```

    //document.addEventListener('deviceready', iniciar);
    iniciar();
}

function iniciar() {
    imgTaco = new Image();
    imgTaco.src = 'img/taco.png';

    imgPonteiro = new Image();
    imgPonteiro.src = 'img/ponteiro.png';

    //somTacada = new Media('file:///android_asset/colisao.mp3');

    iniciarObjetos();
    requestAnimationFrame(animar);
}

```

A função `iniciarObjetos` criará os objetos `taco` e `barraForca`. O controle do taco de bilhar possuirá uma área circular, dentro da qual faremos o gesto de rotação. Já a barra de força será retangular:

```

function iniciarObjetos() {
    taco = new Taco(context, imgTaco);
    taco.raio = 120;
    taco.x = 125;
    taco.y = canvas.height / 2 - 50;

    barraForca = new BarraForca(context, imgPonteiro, taco);
    barraForca.x = taco.x - 100;
    barraForca.y = taco.y + taco.raio + 25;
    barraForca.largura = 200;
    barraForca.altura = 50;
}

```

Por enquanto, o loop de animação será feito através de uma função simples. Basicamente pintamos o Canvas de verde, desenhamos o taco e a barra de força e chamamos o próximo ciclo:

```

function animar() {
    // Fundo verde
    context.save();
}

```

```
context.fillStyle = '#050';
context.fillRect(0, 0, canvas.width, canvas.height);
context.restore();

taco.desenhar();
barraForca.desenhar();
requestAnimationFrame(animar);
}
```

Para podermos testar no Console sem ver nada faltando, já crie o esqueleto da classe `Taco` no arquivo `taco.js`:

```
function Taco(context, imagem) {
    this.context = context;
    this.imagem = imagem;
    this.raio = 0;
    this.x = 0;
    this.y = 0;
    this.rotacao = 0;
    this.forca = 0;
}

Taco.prototype = {
    desenhar: function() {

    }
}
```

E o da classe `BarraForca` em `barra-forca.js`:

```
function BarraForca(context, ponteiro, taco) {
    this.context = context;
    this.ponteiro = ponteiro;
    this.taco = taco;
    this.x = 0;
    this.y = 0;
    this.largura = 0;
    this.altura = 0;
    this.forca = 0;
}

BarraForca.prototype = {
    desenhar: function() {
```

```
    }
}
```

Até aqui, teste no navegador. O único erro deve ser a falta do `cordova.js`.

Vamos desenhar a barra de força. Veja na figura *Taco de bilhar touch*, no começo do capítulo, que ela possui dois retângulos: um interno, representando a força atual, e um externo que delimita a área. O externo é desenhado na largura total da barra, saindo um pouco fora de seus limites para não encobrir o outro. Já o interno é desenhado de acordo com a força atualmente setada no objeto — multipliquei essa força por 2 para a barra não ficar muito pequena. A posição do ponteiro é calculada por outro método, `posicaoPonteiro`, já pensando que vamos precisar dessa posição ao detectar o *pan* sobre ele:

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

  // Contêiner
  ctx.strokeRect(this.x - 2, this.y - 2, this.largura + 4,
    this.altura + 4);

  // Força
  ctx.save();
  ctx.fillStyle = '#900';
  ctx.fillRect(this.x, this.y, this.forca * 2, this.altura);
  ctx.restore();

  // Ponteiro
  var pos = this.posicaoPonteiro();
  ctx.drawImage(this.ponteiro, pos.x, pos.y,
    this.ponteiro.width, this.ponteiro.height);
}, // Atenção para a vírgula
```

Em `posicaoPonteiro`, precisamos colocar a ponta da seta, que horizontalmente está no meio da imagem, na mesma posição *x*

que o final da barra de força. Por isso multiplicamos a força por 2 e descontamos metade da largura da imagem. Já a posição *y* é na metade da barra, mas nada impede que seja um pouco acima ou abaixo:

```
posicaoPonteiro: function() {
    var x = this.x + this.forca * 2 - this.ponteiro.width / 2;
    var y = this.y + this.altura / 2;
    return {x:x, y:y};
}
```

Atribua diferentes valores ao atributo `forca`, no construtor, e vá testando no browser. A barra interna e o ponteiro deverão se ajustar. Volte depois o valor padrão para zero.

Vamos agora detectar o toque no ponteiro. Em `iniciarObjetos`, inicie o Hammer.js e já o passe tanto para o taco quanto para a barra :

```
function iniciarObjetos() {
    var hammer = new Hammer(canvas);

    taco = new Taco(context, imgTaco, hammer);
    // ...

    barraForca =
        new BarraForca(context, imgPonteiro, taco, hammer);
    // ...
}
```

Faça o construtor da `BarraForca` recebê-lo e detectar os eventos de *pan*. Não fizemos o ajuste da direção porque queremos movimentar somente na horizontal, que é o padrão permitido pelo evento:

```
function BarraForca(context, ponteiro, taco, hammer) {
    // ...

    var barra = this;
```

```
hammer.get('pan').set({ threshold: 0 });

hammer.on('panstart', function(e) {
    barra.iniciarAjusteForca(e);
});

hammer.on('panmove', function(e) {
    barra.ajustarForca(e);
});

hammer.on('panend', function(e) {
    barra.finalizarAjusteForca(e);
});

}

}
```

No `panstart`, chamamos o método `iniciarAjusteForca`. Como nosso Canvas é responsivo, temos primeiro de converter a posição do toque em coordenadas da imagem processada em memória, como vimos na seção *Toque em Canvas responsivo*. Quem fará isto é o método `converterParaCanvas`:

```
posicaoPonteiro: function() {
    // ...
}, // Não esqueça da vírgula quando for acrescentar métodos!

iniciarAjusteForca: function(evento) {
    var canvas = this.context.canvas;

    // Coordenadas na imagem do Canvas
    var toque = this.converterParaCanvas(
        evento.center.x - canvas.offsetLeft,
        evento.center.y - canvas.offsetTop
    );

    // continua...
},
```

Em seguida, verificamos se a posição tocada se encontra nos limites do ponteiro. Se o ponteiro foi tocado, sinalizamos que estamos mexendo na força e já chamamos o método que faz o ajuste:

```
iniciarAjusteForca: function(evento) {
```

```

// ...

// Posição do ponteiro
var pos = this.posicaoPonteiro();
var largura = this.ponteiro.width;
var altura = this.ponteiro.height;

// Testar os limites
if (toque.x >= pos.x && toque.x <= pos.x + largura &&
    toque.y >= pos.y && toque.y <= pos.y + altura) {

    this.ajustandoForca = true;
    this.ajustarForca(evento);
}
}, // Ainda tem mais métodos...

```

Somente relembrando, `converterParaCanvas` faz uma regra de três correspondendo às dimensões em tela do Canvas (`offsets`) com as suas dimensões de imagem "real":

```

converterParaCanvas: function(x, y) {
    var canvas = this.context.canvas;

    return {
        x: canvas.width * x / canvas.offsetWidth,
        y: canvas.height * y / canvas.offsetHeight
    }
},

```

Note que o método `ajustarForca`, além de ser o tratador do evento `panmove`, foi também chamado no `panstart` para iniciar o movimento. Testamos se um movimento válido (dentro dos limites) foi iniciado, convertemos a posição para o Canvas e setamos a força de acordo com a posição dentro da barra. Observe que dividimos a nova medida por 2, pois a força é mostrada em tela com o dobro do tamanho. Também limitamos a força entre zero e 100, e setamos a força no objeto `Taco` associado:

```

ajustarForca: function(evento) {
    // Está dentro do limite?

```

```

if (this.ajustandoForca) {
    var canvas = this.context.canvas;

    // Converter para Canvas
    var toque = this.converterParaCanvas(
        evento.center.x - canvas.offsetLeft,
        evento.center.y - canvas.offsetTop
    );

    // Nova força
    this.forca = (toque.x - this.x) / 2;

    // Limites
    if (this.forca < 0) this.forca = 0;
    if (this.forca > 100) this.forca = 100;

    // Força do taco
    this.taco.forca = this.forca;
}
},

```

No evento panend , o método finalizarAjusteForca apenas realiza o último movimento no ponteiro e sinaliza o fim da movimentação:

```

finalizarAjusteForca: function(evento) {
    if (this.ajustandoForca) {
        this.ajustarForca(evento);
        this.ajustandoForca = false;
    }
} // Esta classe acabou!

```

Faça o teste, tanto no browser como agora no aparelho móvel. Se você fez tudo direitinho, já é possível mexer o ponteiro dentro do limite da barra, tanto com o mouse quanto na tela touch.

5.3 ROTACIONANDO UM TACO DE BILHAR

Nesta seção, nos divertiremos um pouco com o evento rotate do Hammer.js. Forneceremos uma área circular, onde o

jogador poderá definir o ângulo do taco realizando uma rotação com dois dedos.

Primeiro, em `taco.js`, no método `desenhar` da classe, desenhe essa área e o taco. Este possui 15 pixels de largura, portanto, descontamos 7 pixels da posição `x` para ficar aproximadamente no meio. Na posição `y`, somamos a força a ser aplicada: quanto maior a força, maior a distância:

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

    // Área de controle
    ctx.beginPath();
    ctx.arc(this.x, this.y, this.raio, 0, Math.PI*2);
    ctx.stroke();

    // Taco
    ctx.drawImage(this.imagem, this.x - 7,
                  this.y + this.forca,
                  this.imagem.width,
                  this.imagem.height);
}
```

Já é possível controlar a força pela barra: o taco se move junto, experimente! Isto ocorre porque, ao movimentar a barra de força, esta seta o atributo `forca` do taco.

Para desenhar o taco rotacionado, podemos usar o método `rotate` do contexto gráfico, que rotaciona o Canvas. Considerando que queremos trabalhar em graus, precisamos converter a rotação para radianos. É muito importante fazermos uso do `save` e do `restore` para desfazer a rotação assim que o taco for desenhado:

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

```

// Área de controle
ctx.beginPath();
ctx.arc(this.x, this.y, this.raio, 0, Math.PI*2);
ctx.stroke();

// Taco
var radianos = this.rotacao * Math.PI / 180;
ctx.save();
ctx.rotate(radianos);
ctx.drawImage(this.imagem, this.x - 7, this.y + this.forca,
    this.imagem.width, this.imagem.height);
ctx.restore();
}

```

Experimente testar, no construtor, valores diferentes no atributo `rotacao`. O taco fica meio deslocado! Se o valor for muito alto, ele chega a sair da tela. Isso acontece porque o centro de rotação do Canvas é seu ponto de origem (0, 0):

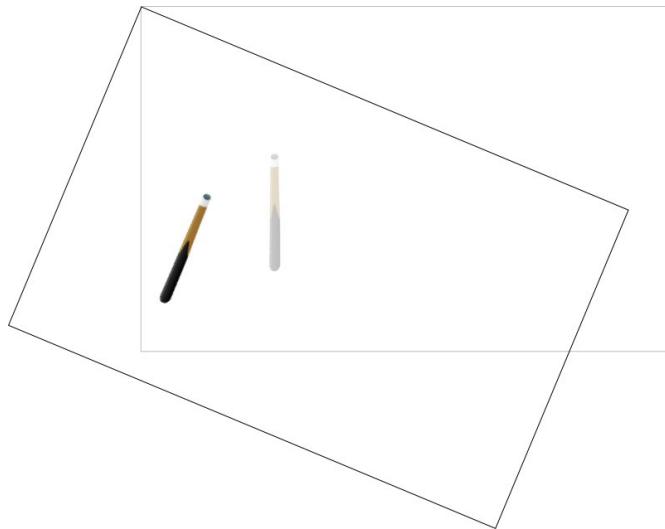


Figura 5.4: O Canvas rotaciona pendurado no ponto (0, 0)

Felizmente, também temos o método `translate`, que desloca

o ponto (0, 0) para onde quisermos. Colocando a origem do Canvas na posição do taco, podemos então rotacioná-lo. Repare nos parâmetros do `drawImage` : com o Canvas transladado, é a partir do ponto (0, 0) que informamos o deslocamento do taco:

```
// Taco
var radianos = this.rotacao * Math.PI / 180;
ctx.save();
ctx.translate(this.x, this.y);
ctx.rotate(radianos);
ctx.drawImage(this.imagem, -7, this.forca,
    this.imagem.width, this.imagem.height);
ctx.restore();
```



Figura 5.5: Deslocando a origem do Canvas, rotacionamos e fazemos o desenho a partir de (0, 0)

Capturando o evento rotate

Não se esqueça de resetar o atributo `rotacao` para zero. A rotação será definida pela captura do evento `rotate` do Hammer.js. No construtor, configure os eventos à semelhança do que fizemos com a barra de força:

```
function Taco(context, imagem, hammer) {
    // ...
```

```

var taco = this;
hammer.get('rotate').set({ enable: true, threshold: 0 });

hammer.on('rotatestart', function(e) {
    taco.iniciarRotacao(e);
});
hammer.on('rotatemove', function(e) {
    taco.rotacionar(e);
});
hammer.on('rotateend', function(e) {
    taco.finalizarRotacao(e);
});
}

```

Também semelhantemente à outra classe, vamos converter as posições para o Canvas em `iniciarRotacao`. Só que agora são dois dedos. Nossa objetivo é saber se os pontos tocados estão na área delimitada. Para isso, chamamos o método `naArea`: se este retornar verdadeiro para os dois dedos, sinalizamos o início da rotação:

```

// Atenção para as vírgulas!
iniciarRotacao: function(evento) {
    var canvas = this.context.canvas;

    var dedo1 = this.converterParaCanvas(
        evento.pointers[0].pageX - canvas.offsetLeft,
        evento.pointers[0].pageY - canvas.offsetTop
    );

    var dedo2 = this.converterParaCanvas(
        evento.pointers[1].pageX - canvas.offsetLeft,
        evento.pointers[1].pageY - canvas.offsetTop
    );

    if (this.naArea(dedo1.x, dedo1.y) &&
        this.naArea(dedo2.x, dedo2.y)) {

        this.rotacionando = true;
        this.rotacaoInicial = this.rotacao;
        this.rotacionar(evento);
    }
}

```

```
},
```

O método converterParaCanvas é o mesmo:

```
converterParaCanvas: function(x, y) {
    var canvas = this.context.canvas;

    return {
        x: canvas.width * x / canvas.offsetWidth,
        y: canvas.height * y / canvas.offsetHeight
    }
},
```

O método naArea verifica, pelo Teorema de Pitágoras, se um ponto pertence a uma área circular (figura *Triângulo retângulo formado com a distância entre o ponto tocado e o círculo*, capítulo *Eventos touch*):

```
naArea: function(x, y) {
    var distanciaX = Math.abs(this.x - x);
    var distanciaY = Math.abs(this.y - y);

    // Pitágoras
    if (distanciaX * distanciaX + distanciaY * distanciaY <=
        this.raio * this.raio)
        return true;
    else
        return false;
},
```

A rotação continua: o método rotacionar obtém o atributo rotation do evento, que é dado em graus. Não podemos ir apenas somando os valores pois, a cada rotatemove que chama este método, aumentamos a rotação naquela quantidade de graus. Em vez disso, sempre partimos da rotação inicial. Também, para simplificar, mantemos o valor dentro do limite de 360 graus:

```
rotacionar: function(evento) {
    if (this.rotacionando) {
        var rotacao = this.rotacaoInicial + evento.rotation;
```

```
this.rotacao = rotacao % 360;  
}  
},
```

ARCOS CÔNGRUOS

Existe arco de 2100° ? Sim, isto significa dar 5 voltas na circunferência e andar mais 300° :

$$\begin{array}{r} 2100 \\ \hline 360 \\ 300 \quad 5 \end{array}$$

Figura 5.6: 2100° e 300° são côngruos, pois param no mesmo ponto

Se a nova rotação do taco de bilhar passar de 360° , o resto da sua divisão por 360 é um arco equivalente. Por isso usamos o operador módulo `%` do JavaScript, que devolve o resto de uma divisão de inteiros.

O método `finalizarRotacao` apenas faz o movimento final e sinaliza que a rotação terminou:

```
finalizarRotacao: function(evento) {  
    if (this.rotacionando) {  
        this.rotacionar(evento);  
        this.rotacionando = false;  
    }  
}
```

Tente fazer o movimento de rotação com 2 dedos dentro da área, e também controle a força da tacada. Você pode achar que ainda não está muito fácil rotacionar o taco. Vejamos por quê.

Capturando dois ou mais gestos juntos

É difícil fazer um *rotate* puro com os dedos, de forma que o Hammer.js detecte o gesto exatamente como um *rotate*. Com um pouco de treino, até se "pega o jeito", mas não é exatamente isso que um jogador frustrado pode pensar. A razão é que qualquer deslocamento mínimo dos dedos pode ser interpretado como um *pinch*.

Podemos tornar o movimento mais suave solicitando que os dois gestos sejam reconhecidos juntos. No construtor, antes de atribuir os tratadores de evento, obtenha referências ao `pinch` e ao `rotate` e, em seguida, chame o método `recognizeWith` em um deles, passando o outro como parâmetro:

```
function Taco(context, imagem, hammer) {
    // ...

    // Configuração dos gestos para serem reconhecidos juntos
    var pinch = hammer.get('pinch');
    var rotate = hammer.get('rotate');
    hammer.get('pinch').set({ enable: true, threshold: 0 });
    hammer.get('rotate').set({ enable: true, threshold: 0 });
    pinch.recognizeWith(rotate);

    // Eventos
    var taco = this;
    hammer.on('rotatestart', function(e) {
        taco.iniciarRotacao(e);
    });

    // ...
}
```

Experimente novamente. Ficou mais fácil, não?

5.4 ANIMANDO A TACADA

Vamos disparar o taco com toda a força! No `index.html`, acrescente uma imagem após a tag `<canvas>`. Já há uma imagem no pacote, mas você pode escolher uma de seu gosto:

```

```

No CSS, posicione-a onde quiser:

```
#tacada {  
    position: absolute;  
    right: 20px;  
    top: 20px;  
}
```

Na função `iniciarObjetos`, vamos programar o `touchstart` da imagem. No callback, preparamos o som e iniciamos a tacada, com duração de 50ms:

```
function iniciarObjetos() {  
    // ...  
  
    var imgTacada = document.getElementById('tacada');  
    imgTacada.addEventListener('touchstart', function() {  
        somTacada.seekTo(0);  
        taco.darTacada(50);  
    });  
}
```

Não esqueça de descomentar a linha que instancia o som na função `iniciar`:

```
somTacada = new Media('file:///android_asset/colisao.mp3');
```

De volta à classe `Taco`, o método `darTacada` vai iniciar indiretamente uma animação, apenas sinalizando que a tacada está ocorrendo. Por isso, guardamos o instante em que a tacada começou, para calcular a posição do taco no decorrer da animação:

```
darTacada: function(duracao) {  
    this.tacada = true;
```

```
this.duracaoTacada = duracao;
this.inicioTacada = Date.now();
},
```

Podemos disparar animações apenas setando um atributo de um objeto, se em algum lugar no seu ciclo de animação ele se encarregar de ler esse atributo. Ao desenhar a imagem, em vez de deslocar o taco em y pelo montante da força, vamos chamar um método, `deslocamento`, que verificará se a animação da tacada está ocorrendo:

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

ctx.drawImage(this.imagem, -7, this.deslocamento(),
  this.imagem.width, this.imagem.height);

// ...
},
```

Nesse novo método, primeiro verificamos se a animação está ocorrendo. Caso contrário, usamos a força para deslocar o taco:

```
deslocamento: function() {
  // Não foi disparada tacada
  if (! this.tacada) return this.forca;

  // continua ...
}
```

Continuando, para calcular o deslocamento, temos de saber quanto tempo se passou desde o início da animação. Se a força é 100 e quero deslocar em 50ms, dá 2 unidades de força por milissegundo ($100/50$). Multiplico isso pelo tempo decorrido e sei quanto o taco vai se deslocar naquele quadro:

```
// Tempo decorrido
var agora = Date.now();
var decorrido = agora - this.inicioTacada;
```

```
// Deslocamento acumulado  
var distancia = this.forca / this.duracaoTacada * decorrido;
```

Depois, verificamos se o tempo excedeu. Em caso positivo, podemos deslocar o taco no montante exato da força e finalizar a animação. A nova posição do taco será a força (afastamento máximo) menos o deslocamento atual (quanto o taco volta em direção ao centro do círculo):

```
// Completou  
if (decorrido >= this.duracaoTacada) {  
    distancia = this.forca;  
    this.tacada = false;  
}  
  
return this.forca - distancia;
```

Quando um evento se encerra, podemos receber uma notificação para começar outro. Por exemplo, quando a tacada terminar, o jogo pode receber um aviso para começar a animação das bolas na mesa:

```
// Completou  
if (decorrido >= this.duracaoTacada) {  
    distancia = this.forca;  
    this.tacada = false;  
  
    // Notificar o jogo  
    if (this.aposTacada) this.aposTacada();  
}
```

No `index.html`, passamos um callback para a tacada:

```
function iniciarObjetos() {  
    // ...  
  
    taco.aposTacada = tacada;  
}
```

E criamos esse callback. Aqui, vamos apenas reproduzir o som:

```
function tacada() {  
    somTacada.play();  
}
```

Também convém impedir uma tacada de começar enquanto outra estiver em andamento:

```
darTacada: function(duracao) {  
    if (this.tacada) return; // Já está em uma tacada  
    this.tacada = true;  
    this.duracaoTacada = duracao;  
    this.inicioTacada = Date.now();  
},
```

Tudo pronto, pode testar no aparelho móvel! Guarde o código deste capítulo com carinho, pois será reutilizado quando formos dar sequência ao jogo.

E aqui finalizamos a primeira parte do livro, sobre plataformas móveis! Na próxima parte, trataremos de física e a biblioteca *Box2dWeb*, essencial para quem pretende levar a sério a área de games. Ela será responsável por toda a ação em cima da mesa de bilhar.

Física

CAPÍTULO 6

INTRODUÇÃO AO BOX2DWEB

Façamos um rápido exercício mental. Com o que temos, o que é necessário para fazer um herói pular?

Basicamente, deslocá-lo para cima por um instante, e depois, de volta para baixo. Mas claro que não é só isso. Se, na subida, ele se chocar com o teto ou uma plataforma, o movimento deve se inverter antecipadamente. Na descida, a colisão com o chão é responsável por parar o movimento. Fora que podemos ter animações de sprite diferentes para a subida e para a descida.

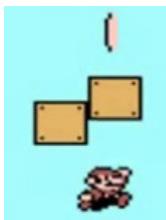


Figura 6.1: Uma colisão alterando o sentido de movimento de um sprite

Por mais complexo que pareça, isso pode ser feito de maneira relativamente simples se soubermos definir bem os **estados** do sprite: dividir seus movimentos em etapas (os tais estados), sendo

que em cada uma ele se comporta e se desenha de maneira diferente, e gerenciar as **transições de estado**, ou seja, detectar momentos quando o sprite passa de um estado a outro. Caso queira saber mais sobre gerenciamento de estados, falei sobre isso em meu livro anterior.

Um recurso muito usado, no entanto, é o uso de **engines físicas**, ou motores de física. São bibliotecas, frameworks, que pretendem simular física realista em um computador, poupando o programador de conhecer fórmulas e inúmeros detalhes. No mundo físico, nosso herói é simplesmente um **corpo**, no qual aplicamos uma **força** de baixo para cima, fazendo-o saltar. A **gravidade** se encarrega de fazê-lo voltar para o chão, e a colisão com este é tratada automaticamente pelo engine, que o faz parar, ou até mesmo quicar, se desejarmos.

Esta é a proposta do **Box2D**, biblioteca desenvolvida em C++ por Erin Catto, programador na Blizzard Entertainment. O primeiro protótipo foi criado como uma demonstração na Game Developers Conference, em 2006, e desde então a biblioteca foi portada para várias outras linguagens, tamanha foi sua aceitação. A API é praticamente idêntica em C++, Java, C#, Lua, ActionScript e JavaScript. Para esta última, vamos usar o port chamado **Box2dWeb**, convertido a partir da *Box2DFlash* desenvolvida em ActionScript.

DOWNLOAD

Faça o download da Box2dWeb em:
<https://code.google.com/p/box2dweb/downloads/list>

Existe outro port da Box2D para JavaScript, o *Box2DJS*. Ele não é atualizado há bastante tempo e requer outras bibliotecas, mas acho interessante mencioná-lo porque na sua homepage você pode ver exemplos da engine em ação:

<http://box2d-js.sourceforge.net/>

Convenceu-se? Um leque de possibilidades abre-se para nós a partir de agora.

6.1 UM PRIMEIRO TUTORIAL

Embora não seja uma engine difícil de usar, a Box2D ou seus ports requerem alguma prática e podem assustar o programador iniciante. A API é composta de inúmeros objetos, e é fundamental saber o papel de cada um na engine.

A intenção deste capítulo é mostrar-lhe uma teoria básica, acompanhada de sua aplicação prática. Trabalharemos com o mundo físico, gravidade, corpos, *fixtures* e formas.

Este exercício pode ser feito como uma simples página web, embora no pacote de download esteja configurado como um projeto Cordova, de nome `box2dweb-intro`. Segue o esqueleto da `index.html`. O Canvas possui tamanho fixo, pois não

queremos neste momento poluir o código com outras preocupações que não sejam sobre a Box2dWeb:

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,
        height=device-height, user-scalable=no,
        initial-scale=1, maximum-scale=1, minimum-scale=1">
    <title>Introdução ao Box2dWeb</title>

    <script src="cordova.js"></script>
    <script src="rAF.js"></script>
    <script src="Box2dWeb-2.1.a.3.min.js"></script>
    <script>
        // Inicialização aqui
    </script>
</head>

<body>
    <canvas id="box2dweb" width="640" height="480"></canvas>
</body>

</html>
```

A inicialização não é diferente do que já vimos:

```
var canvas, context;

window.onload = function() {
    canvas = document.getElementById('box2dweb');
    context = canvas.getContext('2d');
    //document.addEventListener('deviceready', iniciar);
    iniciar();
}

function iniciar() {
    // Aqui começará a física realista!
}
```

Esqueleto pronto, vamos entrar nos conceitos da engine, uma

etapa por vez.

6.2 OBJETOS FUNDAMENTAIS

Mundo (b2World)

O primeiro objeto criado é sempre o *mundo*, que representa o espaço onde os corpos interagem. Normalmente, só existe um mundo físico por jogo, pois a necessidade de processamento é elevada.



Figura 6.2: O mundo físico em um jogo

É preciso ressaltar que esse mundo existe apenas na memória do computador. A cada quadro de animação, nós avançamos o mundo físico no tempo e a engine recalcula as posições dos objetos dentro dele, detecta colisões, entre outras coisas, de acordo com as forças que estiverem atuando em cada corpo. Cabe a nós obter as posições de todos os corpos e desenhar os sprites.

A Box2dWeb preserva o sistema de espaços de nomes presente na engine original, na forma de objetos que agrupam outros objetos. Por exemplo, para criar um `b2World`, teríamos de referenciar o construtor `Box2D.Dynamics.b2World`:

```
// Exemplo teórico
var mundo = new Box2D.Dynamics.b2World( ... );
```

Para reduzir a complicaçāo, o comum é obter referências a esses construtores no início do código. Posteriormente, usamos essa referência para instanciar os objetos da engine:

```
// Construtores
var b2World = Box2D.Dynamics.b2World;

// Variáveis do jogo
var canvas, context, mundo;

// ...

// Criando o mundo
function iniciar() {
    mundo = new b2World( ... ); // O que passamos aqui?
}
```

O construtor `b2World` requer os parâmetros:

- um objeto `b2Vec2` representando a gravidade (veja logo a seguir);
- um booleano (`true` / `false`) indicando se os corpos podem *dormir* durante a simulação.

Um corpo "dorme" quando está parado e não há forças atuando sobre ele, e a engine o "acorda" automaticamente se isso acontecer. Um corpo dormindo é ignorado pela engine durante esse estado, o que economiza processamento.

Vetor (`b2Vec2`)

Para podermos criar um mundo físico, precisamos passar para ele um vetor representando a gravidade. Em Física, um **vetor** é toda grandeza que possui direção e sentido. Por exemplo, a

gravidade possui direção vertical e sentido para baixo. A direção de um vetor pode ser diagonal, mas o mais comum é pensarmos separadamente nos componentes horizontal (x) e vertical (y).

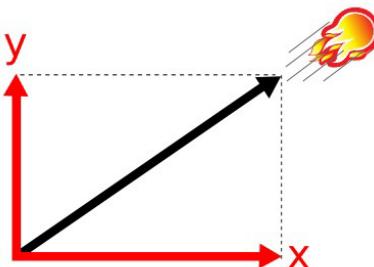


Figura 6.3: Uma trajetória é um vetor, portanto pode ser decomposta em x e y

A gravidade atua somente para baixo, com determinado valor y , enquanto o valor x é zero. A gravidade da Terra é de 9,81 metros por segundo ao quadrado, ou seja, a velocidade de um corpo em queda livre, sem a resistência do ar, aumenta 9,81 metros por segundo **a cada segundo**.

A Box2D e seus ports representam grandezas vetoriais através do objeto `b2Vec2`. Seu construtor recebe os valores nos eixos x e y . Para criar um mundo com essa gravidade, permitindo que os corpos durmam, temos de fazer:

```
// Construtores
var b2Vec2 = Box2D.Common.Math.b2Vec2;
var b2World = Box2D.Dynamics.b2World;

// ...

function iniciar() {
    // Criar o mundo
    var gravidade = new b2Vec2(0, 9.81); // x, y
    mundo = new b2World(gravidade, true); // Corpos podem dormir
```

```
// continua ...  
}
```

Definições de corpos e fixtures (b2BodyDef e b2FixtureDef)

Dentro do mundo, o *corpo* representa um objeto indivisível sujeito à ação de forças. Uma bola, um homem, uma pedra são corpos no mundo físico.

Um corpo é formado por uma ou mais *fixtures*, que são suas unidades materiais. Cada fixture possui uma forma e propriedades como densidade, atrito e elasticidade.

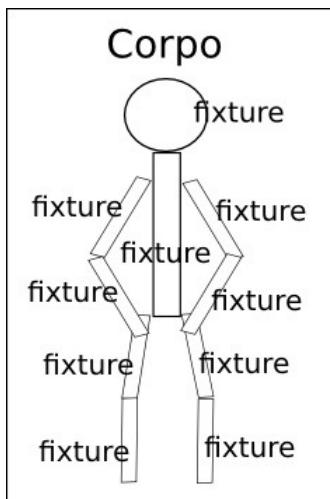


Figura 6.4: Corpo físico e suas unidades (fixtures)

Fixture: fixação?

fixture *fix.ture* 1 fixação, fixidez. 2 pessoa ou coisa permanentemente ligada a um lugar. 3 acessório, pertence, instalação. [...]

Fonte: <http://michaelis.uol.com.br/moderno/ingles/>.

Uma "fixture" pode ser entendida como algo que é fixo em uma estrutura, como uma peça de um carro ou um relógio pendurado na parede: ambos ficam imóveis na estrutura do carro ou da construção. Muitas traduções são dadas para a palavra, como "fixação" ou "acessório", mas nenhuma me soa bem em português, razão por que opto em usar o termo em inglês: *fixture*.

Esses elementos são representados na engine por `b2Body` e `b2Fixture`, mas nós não os instanciamos diretamente. O que criamos são objetos de definição, que contêm as informações usadas pela engine para criar os verdadeiros corpos e fixtures. Os objetos de definição são `b2BodyDef` e `b2FixtureDef`:

```
// Construtores  
// ...  
var b2BodyDef = Box2D.Dynamics.b2BodyDef;  
var b2FixtureDef = Box2D.Dynamics.b2FixtureDef;
```

Para criar uma definição de corpo, o mínimo necessário é definir sua posição (x, y) em metros. Para passar valores em pixels, divida-os por uma escala:

```
// Variáveis do jogo
```

```
// ...
var escala = 30; // pixels por metro

function iniciar() {
    // ...

    // Definição de corpo
    var corpoDef = new b2BodyDef();
    corpoDef.position.Set(200/escala, 100/escala);

    // continua ...
}
```

MEDIDAS NO Box2D

Um detalhe muito importante é o uso das unidades **metro**, **quilograma** e **segundo** nas simulações físicas. Nossa atenção se voltará principalmente para os comprimentos, que sempre devem ser convertidos de pixels para metros, ao configurar os objetos, e de metros para pixels novamente, ao ler suas posições. Isso não é difícil de se fazer: basta definirmos uma escala , um número de pixels por metro a ser usado na simulação.

```
var escala = 30;
```

Ao criar um objeto, dividimos o tamanho em pixels desejado pela escala. Um exemplo simples: se a escala for de 100 pixels por metro, um objeto de 50 pixels terá meio metro (50/100).

```
var larguraMetros = 25 / escala; // 25 pixels
```

Ao obter a posição ou tamanho de um objeto em metros, fazemos o processo inverso: multiplicamos o valor pela escala para obter em pixels:

```
var larguraMetros = ...;
var larguraPixels = larguraMetros * escala;
```

Para dar formato ao corpo, criamos uma ou mais fixtures. O mínimo necessário é dar à fixture uma *forma*, que corresponde a uma figura geométrica simples. No próximo capítulo, veremos as opções de formas existentes. Combinando várias fixtures, podemos obter um corpo com formato mais complexo. Aqui, criamos uma forma circular de 25 pixels de raio:

```
function iniciar() {  
    // ...  
  
    // Definição de fixture  
    var fixtureDef = new b2FixtureDef();  
    fixtureDef.shape = new b2CircleShape(25/escala);  
  
    // continua...  
}
```

Claro, acrescente a referência ao construtor `b2CircleShape` no início do código:

```
// Construtores  
// ...  
var b2CircleShape = Box2D.Collision.Shapes.b2CircleShape;
```

Por fim, temos de criar o corpo no mundo e criar nele a fixture:

```
function iniciar() {  
    // ...  
  
    // Criar o corpo  
    var bola = mundo.CreateBody(corpoDef);  
    bola.CreateFixture(fixtureDef);  
  
    // continua...  
}
```

Renderização gráfica de debug (`b2DebugDraw`)

Como dito anteriormente, o mundo físico existe somente em memória. A engine física não realiza tarefas de renderização. Não podemos simplesmente passar imagens para ela e deixar que ela cuide disso para nós.

No entanto, a Box2D original oferece a possibilidade de se implementar um *Debug Draw*: um renderizador simples, voltado para testes, que nos permitiria acompanhar o que está acontecendo

no mundo físico. Certamente, há inúmeros deles implementados por aí, para cada port da engine. A **Box2dWeb** já vem com um **Debug Draw** pronto para uso, herdado do Box2DFlash.

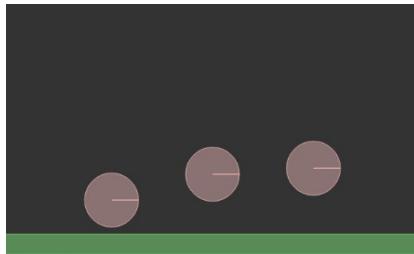


Figura 6.5: Debug Draw renderizando o mundo físico com desenhos simples

As vantagens de se usar o Debug Draw são:

- poder trabalhar no desenvolvimento enquanto a equipe de arte trabalha nos gráficos;
- ajuda a localizar melhor a origem de bugs. Se o Debug Draw desenha os objetos nas posições corretas, o erro é na sua lógica de renderização de sprites; caso contrário, é na lógica física.

Ainda na função `iniciar`, a criação de um Debug Draw envolve as configurações a seguir. Destaque para o método `SetFlags`, onde configuramos que componentes da engine queremos desenhar. A variável `e_shapeBit` solicita o desenho das formas associadas às fixtures:

```
function iniciar() {  
    // ...  
  
    // Criar o Debug Draw  
    var debugDraw = new b2DebugDraw();  
    debugDraw.SetSprite(context);      // Contexto gráfico
```

```
    debugDraw.SetDrawScale(escala); // Pixels por metro
    debugDraw.SetFillAlpha(0.6); // Opacidade
    debugDraw.SetLineThickness(1.0); // Espessura das linhas
    debugDraw.SetFlags(b2DebugDraw.e_shapeBit); // Formas

    // Configurá-lo no mundo
    mundo.SetDebugDraw(debugDraw);
}
```

Não se esqueça de adicionar a referência ao construtor `b2DebugDraw` :

```
// Construtores
// ...
var b2DebugDraw = Box2D.Dynamics.b2DebugDraw;
```

Os objetos de nosso mundo estão configurados. Precisamos agora fazer a física acontecer nele.

6.3 ANIMAÇÃO DO MUNDO FÍSICO

Avançando o tempo na animação

Vamos terminar a função `iniciar` disparando a animação. Temos de controlar quanto tempo há entre um ciclo e outro, para saber quanto tempo avançar no mundo físico. Por isso, guardamos a hora atual na variável `ultimoCiclo` :

```
function iniciar() {
    // ...

    // Iniciar a animação
    ultimoCiclo = Date.now();
    requestAnimationFrame(animar);
}
```

A função de animação necessita avançar o mundo físico no tempo. O método `Step` do mundo pede um tempo em segundos,

por isso dividimos o valor que temos (em milissegundos) por 1000. Os valores 8 e 3 representam os números de iterações que o mundo vai realizar para movimentar os objetos e para verificar sobreposições, respectivamente. Quanto maior o número de iterações, maior a acurácia — e menor o desempenho. Esses são os valores recomendados pelo próprio Erin Catto no manual da Box2D, e podem ser usados como padrão caso não tenhamos excesso de objetos e o desempenho seja satisfatório.

Tendo avançado o tempo, chamamos o método `DrawDebugData` do mundo para desenhar as formas no Canvas pelo objeto Debug Draw:

```
function animar() {
    var agora = Date.now();
    var decorrido = agora - ultimoCiclo;

    mundo.Step(decorrido/1000, 8, 3); // Avançar
    mundo.DrawDebugData(); // Gráficos de debug

    ultimoCiclo = agora;
    requestAnimationFrame(animar);
}
```

Rode o projeto. Vemos um círculo estático no Canvas, com uma linha traçada no raio. Essa linha serve para visualizarmos a rotação da forma circular, quando forem aplicadas forças no corpo.



Figura 6.6: Nossa corpo não se move!

Corpos estáticos e dinâmicos

Por padrão, os corpos são **estáticos**, isto é, ficam parados no seu lugar. Servem de barreira para outros corpos se movimentando, mas eles mesmos não reagem a estímulos, como colisões ou aplicação de forças.

Se quisermos que nossa bola caia em função da gravidade, temos de transformá-la em um corpo **dinâmico**. No trecho em que criamos o `b2BodyDef` , vamos setar o atributo `type` como `b2Body.b2_dynamicBody` :

```
// Definição de corpo
var corpoDef = new b2BodyDef();
corpoDef.position.Set(200/escala, 100/escala);
corpoDef.type = b2Body.b2_dynamicBody;
```

E, claro, referenciar `b2Body` :

```
// Construtores
// ...
var b2Body = Box2D.Dynamics.b2Body;
```

Fiz as referências uma a uma, conforme necessário, para você tomar contato com os principais objetos do Box2dWeb. Nossos próximos projetos já podem começar com um conjunto de referências pronto.

Agora a bola cai! Repare que ela mudou de cor: objetos dinâmicos são representados em roxo pelo Debug Draw, enquanto os estáticos eram verdes. Se você achar que ela está caindo muito depressa, diminua o valor `y` do vetor da gravidade e note que ela cairá mais devagar. Faça experiências: coloque um valor negativo e veja-a subir; atribua um valor ao eixo `x` e ela se deslocará para o lado.

DESEMPENHO DA ANIMAÇÃO: REQUESTANIMATIONFRAME X SETTIMEOUT

O Box2dWeb trabalha bem com o velho `setTimeout`, aliás, foi feito para ser usado com ele! Faz pouco tempo que o `requestAnimationFrame` passou a ser adotado em larga escala pelos browsers.

O uso comum é avançarmos 60 quadros por segundo, dividindo 1000 milissegundos por 60:

```
setTimeout(animar, 1000/60);
```

Já o avanço do tempo no Box2dWeb é dado em segundos, ou seja, 1 segundo dividido por 60:

```
mundo.Step(1/60, 8, 3);
```

No próximo capítulo, você aprenderá que aplicando forças a um corpo podemos mudar a sua trajetória. Uma bola pode ser arremessada para cima, subindo até uma altura máxima de acordo com a força aplicada, e a gravidade a fará descer novamente. Se quisermos, podemos parar o movimento em uma altura determinada, e deixá-la à mercê da gravidade.

Se a bola for lançada em sentido diagonal, a engine automaticamente traça uma trajetória em arco. Também podemos movimentar um corpo manualmente, como se fosse a nave ou os OVNIs do capítulo *Incrementando o jogo de nave*.



Figura 6.7: Um objeto dinâmico aparece roxo no Debug Draw

Sugiro que estude bastante estes conceitos básicos, pois eles são a base de tudo o que faremos com o Box2dWeb. No próximo, vamos conhecer todas as formas que podem ser usadas, aprender a configurar propriedades físicas dos objetos e aplicar forças, impulsos e velocidade neles.

DOCUMENTAÇÃO ORIGINAL DA BOX2D

O funcionamento da engine e a sua API foram descritas pelo próprio criador, Erin Catto, em:

<http://box2d.org/manual.pdf>

SUGESTÃO DE LEITURA

Box2D for Flash Games, de Emanuele Feronato (2012).

CAPÍTULO 7

CONFIGURAÇÕES E MOVIMENTO DE CORPOS

No capítulo anterior, aprendemos a criar um mundo físico e as etapas necessárias para colocar um corpo nele. Só esse "básico" já exige de nós conhecer a sequência:

1. criar mundo;
2. criar definição de corpo (`b2BodyDef`);
3. criar definição de fixture (`b2FixtureDef`);
4. criar uma forma e associá-la ao `b2FixtureDef` ;
5. criar o corpo e sua fixture a partir das definições.

Indo um passo além, vamos conhecer mais detalhes e diversas opções que temos em cada um desses passos. Talvez você esteja se perguntando por que precisamos criar fixtures em vez de associar a forma diretamente ao corpo. Veremos que elas são mais do que um simples intermediário entre o corpo e uma forma.

7.1 FORMAS

O corpo que criamos no último exercício tinha forma redonda, criada através de um `b2CircleShape`. Essa forma é atribuída à propriedade `shape` da definição da fixture:

```
fixtureDef.shape = new b2CircleShape(25/escala);
```

Vamos criar uma nova página web. No pacote de download, este projeto se encontra na pasta `formas`, já configurada para o Cordova. Os arquivos de script indicados podem ser obtidos do pacote ou dos projetos anteriores:

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width,
        height=device-height, user-scalable=no,
        initial-scale=1, maximum-scale=1, minimum-scale=1">
    <title>Formas</title>

    <script src="cordova.js"></script>
    <script src="rAF.js"></script>
    <script src="Box2dWeb-2.1.a.3.min.js"></script>
    <script>
        // Inicialização aqui
    </script>
</head>

<body>
    <canvas id="formas" width="640" height="480"></canvas>
</body>

</html>
```

Vamos iniciar o código fazendo referência aos principais construtores e criando o mundo físico:

```
var b2Vec2 = Box2D.Common.Math.b2Vec2;
var b2World = Box2D.Dynamics.b2World;
var b2BodyDef = Box2D.Dynamics.b2BodyDef;
var b2FixtureDef = Box2D.Dynamics.b2FixtureDef;
var b2CircleShape = Box2D.Collision.Shapes.b2CircleShape;
var b2PolygonShape = Box2D.Collision.Shapes.b2PolygonShape;
var b2DebugDraw = Box2D.Dynamics.b2DebugDraw;
var b2Body = Box2D.Dynamics.b2Body;
```

```
var canvas, context, mundo;
var escala = 30; // pixels por metro

window.onload = function() {
    canvas = document.getElementById('formas');
    context = canvas.getContext('2d');
    mundo = new b2World(new b2Vec2(0, 9.81), true);
    debugDraw();
    iniciar();
    //document.addEventListener('deviceready', iniciar);
}
```

Abstraímos para a função `debugDraw` a tarefa de configurar a renderização padrão de debug. Este código pode ser copiado a partir do projeto do capítulo anterior:

```
function debugDraw() {
    var debugDraw = new b2DebugDraw();
    debugDraw.SetSprite(context);
    debugDraw.SetDrawScale(escala);
    debugDraw.setFillAlpha(0.6);
    debugDraw.SetLineThickness(1.0);
    debugDraw.SetFlags(b2DebugDraw.e_shapeBit); // Formas!!
    mundo.SetDebugDraw(debugDraw);
}
```

Retângulos básicos (caixas)

Começaremos criando algumas paredes delimitando a área do Canvas, para que outros corpos não consigam sair. Essas paredes serão corpos estáticos de formato retangular:

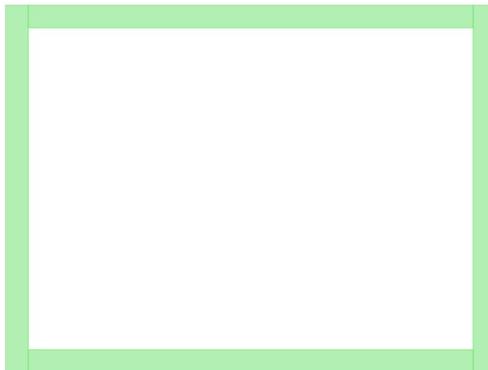


Figura 7.1: Formas retangulares

No Box2D e seus ports, temos dois tipos de retângulos, sendo que ambos são definidos a partir do centro:

- **Caixas:** retângulos básicos, que têm a posição do corpo como centro;
- **Caixas orientadas:** podem ter inclinação e seu centro pode ser deslocado *relativamente* à posição do corpo.

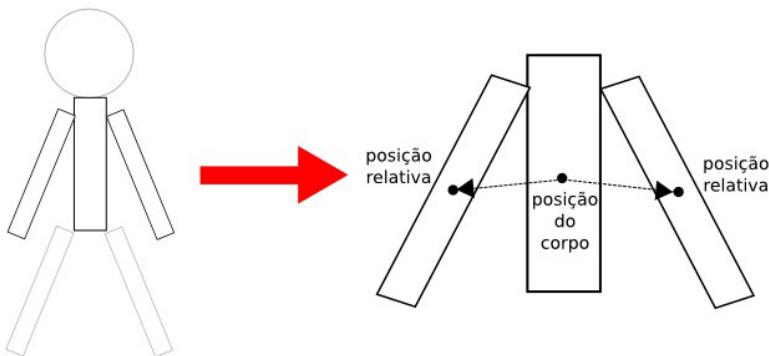


Figura 7.2: Caixa comum (centro) e caixas orientadas inclinadas e deslocadas a partir do centro do corpo

Vamos usar primeiro as caixas comuns, para criar as paredes. Como são quatro, convém criar uma função separada, `parede`, a ser chamada em `iniciar`. Passamos para ela as coordenadas e dimensões que queremos em pixels:

```
function iniciar() {  
    // x, y, largura, altura  
    parede(0, 0, 30, 480);  
    parede(610, 0, 30, 480);  
    parede(30, 0, 580, 30);  
    parede(30, 450, 580, 30);  
  
    // Haverá mais objetos!  
}
```

Podemos começar a função `parede` ajustando os valores conforme a escala definida. Lembre-se de que o mundo físico requer os dados em metros, não em pixels:

```
function parede(x, y, largura, altura) {  
    // Ajustar para escala  
    x /= escala;  
    y /= escala;  
    largura /= escala;  
    altura /= escala;  
  
    // continua...  
}
```

Se estamos recebendo `x` e `y` como o canto superior esquerdo, temos de somar metade da largura e da altura para determinar o centro:

```
function parede(x, y, largura, altura) {  
    // ...  
  
    // Meia caixa  
    var meiaLargura = largura / 2;  
    var meiaAltura = altura / 2;  
  
    var corpoDef = new b2BodyDef();
```

```
corpoDef.position.Set(x + meiaLargura, y + meiaAltura);

// continua...
}
```

Caixas são criadas usando `b2PolygonShape`. O método `SetAsBox` cria uma caixa simples, tomando a posição do corpo como centro. Devemos passar para ele as *metades* da largura e da altura que queremos. Depois disso, basta criar a fixture com a forma e o corpo no mundo:

```
function parede(x, y, largura, altura) {
    // ...

    // Formato retangular
    var forma = new b2PolygonShape();
    forma.SetAsBox(meiaLargura, meiaAltura);

    var fixtureDef = new b2FixtureDef();
    fixtureDef.shape = forma;

    // Criar
    var corpo = mundo.CreateBody(corpoDef);
    corpo.CreateFixture(fixtureDef);
}
```

Para podermos ver os corpos, vamos chamar a função de animação ao fim de `iniciar`:

```
function iniciar() {
    // ...

    ultimoCiclo = Date.now();
    requestAnimationFrame(animar);
    //setTimeout(animar, 1000/60);
}
```

Procure depois fazer experiências tanto com `requestAnimationFrame` quanto com o `setTimeout`. Com poucos corpos, você não notará grandes diferenças, mas no jogo de

bilhar você se surpreenderá ao ver que o `setTimeout` apresentará melhor desempenho.

A função de animação simplesmente avança o tempo decorrido, desenha o Debug Draw e chama o próximo ciclo:

```
function animar() {  
    var agora = Date.now();  
    var decorrido = agora - ultimoCiclo;  
  
    mundo.Step(decorrido/1000, 8, 3);  
    mundo.DrawDebugData();  
  
    ultimoCiclo = agora;  
    requestAnimationFrame(animar);  
    //setTimeout(animar, 1000/60);  
}
```

As paredes já podem ser vistas!

Retângulos orientados

Se só pudéssemos criar retângulos a partir do centro do corpo, ficaríamos de mãos atadas. Para corpos complexos, com várias fixtures, queremos definir a posição de cada uma. O método `SetAsOrientedBox` , do `b2PolygonShape` , permite-nos posicionar a forma relativamente ao centro do corpo e também definir um ângulo de inclinação.

Vamos criar um corpo em forma de "X". Para isso, precisamos de dois retângulos inclinados: um em sentido horário, outro em sentido anti-horário.

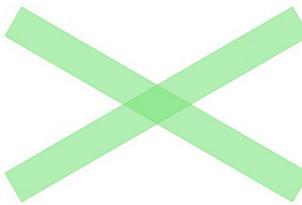


Figura 7.3: Formas retangulares orientadas com inclinação

Em `iniciar`, chame a função `xis`. Pode ser antes ou depois das paredes, não importa. Ele receberá as coordenadas do centro, o comprimento do braço (meia-largura do retângulo), a espessura (meia-altura) e o ângulo de inclinação:

```
function iniciar() {
    // x, y, braço, espessura, inclinação
    xis(320, 240, 200, 20, 30);

    // ...
}
```

Comece a função `xis` criando as caixas retangulares. O método `SetAsOrientedBox` recebe a meia-largura, a meia-altura, a posição **relativa ao centro do corpo** e o ângulo de inclinação. Para a posição, criamos um `b2Vec2` com valores zerados, para colocá-los exatamente no centro do corpo. Repare que um dos ângulos de inclinação deve ter sinal negativo, para inverter o sentido:

```
function xis(x, y, braco, espessura, inclinacao) {
    // Posicionar os retângulos
    var posicao = new b2Vec2(0, 0);
    var angulo = inclinacao * Math.PI / 180;

    var ret1 = new b2PolygonShape();
    ret1.SetAsOrientedBox(braco/escala, espessura/escala,
        posicao, angulo);
```

```
var ret2 = new b2PolygonShape();
ret2.SetAsOrientedBox(braco/escala, espessura/escala,
                      posicao, -angulo);

// continua...
}
```

Por fim, basta criar o corpo. Em caso de corpos com várias fixtures, você pode usar o mesmo `b2FixtureDef`, apenas alterando as propriedades conforme cria cada fixture:

```
function xis(x, y, braco, espessura, inclinacao) {
    // ...

    // Criar corpo e fixtures
    var bodyDef = new b2BodyDef();
    bodyDef.position.Set(x/escala, y/escala);
    var corpo = mundo.CreateBody(bodyDef);

    var fixtureDef = new b2FixtureDef();
    fixtureDef.shape = ret1;
    corpo.CreateFixture(fixtureDef);
    fixtureDef.shape = ret2; // Mudando a forma do b2FixtureDef
    corpo.CreateFixture(fixtureDef);
}
```

O cenário já tem um obstáculo!

Polígonos

Claro que o `b2PolygonShape`, tendo o nome que tem, não serve apenas para criar retângulos. O Box2D e ports permitem a criação de polígonos convexos, sem "aberturas". Polígonos côncavos não podem ser criados diretamente, mas podemos juntar dois ou mais convexos para formar um côncavo:

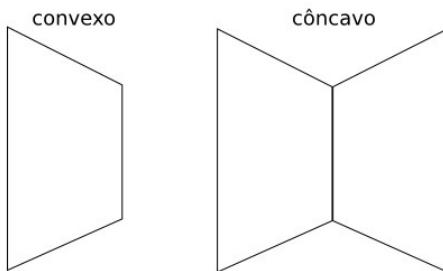


Figura 7.4: Polígonos convexos e côncavos

CONVEXO E CÔNCAVO

Se você, como eu, é daqueles que precisam parar para pensar quando tem de distinguir convexo de côncavo, vai aí uma dica: **convexo** é o lado de trás da colher. O lado que tem a abertura é **côncavo** ("com cavidade").



Figura 7.5: Lados convexo e côncavo da colher

Pense sempre nas costas da colher ao criar um polígono no Box2dWeb.

Vamos criar um triângulo. Em `iniciar`, chame uma nova função, `triângulo`:

```
function iniciar() {  
    triangulo();  
  
    // ...  
}
```

Nessa nova função, precisamos criar um array de pontos que formarão os vértices do polígonos. O Box2dWeb processará esses pontos em **sentido horário**, portanto, caso haja erros na definição de seus polígonos, confira se esses pontos, sendo percorridos dessa forma, formam a figura desejada.

Os valores continuam sendo relativos ao centro do corpo. Esses pontos são passados para o método `SetAsVector`, que também recebe o número de vértices a serem processados:

```
function triangulo() {  
    var pontos = [  
        new b2Vec2(0, -30/escala),  
        new b2Vec2(30/escala, 0),  
        new b2Vec2(-30/escala, 0)  
    ]  
  
    var forma = new b2PolygonShape();  
    forma.SetAsVector(pontos, 3);  
  
    // continua...  
}
```

Proceda à criação do corpo. Para vê-lo caindo, vamos torná-lo dinâmico:

```
function triangulo() {  
    // ...  
  
    // Corpo  
    var corpoDef = new b2BodyDef();
```

```

corpoDef.type = b2Body.b2_dynamicBody;
corpoDef.position.Set(320/escala, 100/escala);

var fixtureDef = new b2FixtureDef();
fixtureDef.shape = forma;

var corpo = mundo.CreateBody(corpoDef);
corpo.CreateFixture(fixtureDef);
}

```

Já é possível ver o corpo triangular caindo sobre o "X" pela ação da gravidade. E o que é mais interessante, a **colisão** entre diferentes formas já é tratada pelo Box2dWeb!

Mas... e se quisermos dar-lhe algum trajeto diferente?

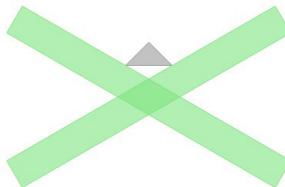


Figura 7.6: Corpo poligonal

7.2 MOVIMENTANDO CORPOS

Existem três formas de movimentar corpos: aplicando forças, aplicando impulsos ou setando a velocidade diretamente. Vamos criar um novo projeto para testar essas três formas, de nome `movimento`. O esqueleto e o script de inicialização do Box2dWeb e do Canvas são os mesmos do projeto anterior (seção *Formas*), por isso não vou repeti-los aqui. Copie-os e também as funções `debugDraw` e `animar`.

Vou começar a explicação pela função `iniciar`, na qual

criaremos uma bolinha e um chão no qual ela pode se apoiar. Precisamos guardar uma referência à bola se quisermos movimentá-la depois.

```
function iniciar() {
    bola = criarBola(320, 450, 15);
    chao(320, 470, 10);

    ultimoCiclo = Date.now();

    // Escolha qual tiver melhor desempenho para SEU jogo
    //requestAnimationFrame(animar);
    setTimeout(animar, 1000/60);
}
```

Adicione a bola às declarações de variáveis da página:

```
var canvas, context, mundo, bola;
var escala = 30; // pixels por metro
```

A função criarBola não tem novidades: ela cria a forma circular, as definições e o corpo a partir delas. O corpo criado é retornado para que a referência possa ser guardada:

```
function criarBola(x, y, raio) {
    var forma = new b2CircleShape(raio/escala);

    var fixtureDef = new b2FixtureDef();
    fixtureDef.shape = forma;

    var corpoDef = new b2BodyDef();
    corpoDef.type = b2Body.b2_dynamicBody;
    corpoDef.position.Set(x/escala, y/escala);

    var corpo = mundo.CreateBody(corpoDef);
    corpo.CreateFixture(fixtureDef);

    return corpo;
}
```

Para a função chao não ficar muito confusa, já recebemos as

coordenadas como o centro do retângulo. A largura será a mesma do Canvas e a altura é o parâmetro `espessura`. Estes valores devem ser divididos por 2, pois o `SetAsBox` recebe meia-largura e meia-altura:

```
function chao(x, y, espessura) {
    var forma = new b2PolygonShape();
    forma.SetAsBox(canvas.width/2/escala, espessura/2/escala);

    var fixtureDef = new b2FixtureDef();
    fixtureDef.shape = forma;

    var corpoDef = new b2BodyDef();
    corpoDef.position.Set(x/escala, y/escala);

    corpo = mundo.CreateBody(corpoDef);
    corpo.CreateFixture(fixtureDef);
}
```

Já podemos ver a bola sobre o chão:



Figura 7.7: Bola caindo no chão

Aplicando forças

Vamos, a cada quadro da animação, aplicar uma força na bola, fazendo-a subir. Para isso, vamos obter seu centro de massa, dado pelo método `GetWorldCenter`, e aplicar a força nesse ponto, com o método `ApplyForce`:

```
function animar() {
    var centro = bola.GetWorldCenter();
    bola.ApplyForce(new b2Vec2(0, -5), centro);

    // ...
}
```

A bola faz um movimento acelerado para cima, como um foguete! Uma força deve ser aplicada continuamente, por isso nós a aplicamos dentro do loop de animação.

Aplicando impulsos

Se aplicarmos a força apenas uma vez, seu efeito será inócuo. No entanto, podemos aplicar um impulso uma única vez, e deixar que a gravidade atue a partir daí.

Retire ou comente as linhas que aplicam a força em `animar`. Agora, vamos aplicar um impulso único em `iniciar`:

```
function iniciar() {  
    // ...  
  
    var centro = bola.GetWorldCenter();  
    bola.ApplyImpulse(new b2Vec2(3, -10), centro);  
}
```

Com valores x e y no vetor, a bola dá um salto para o lado!

Alterando a velocidade

Podemos também setar diretamente a velocidade da bola com o método `SetLinearVelocity`, sem depender do centro de massa:

```
function iniciar() {  
    // ...  
  
    bola.SetLinearVelocity(new b2Vec2(10, -200));  
}
```

Movimentando arbitrariamente

A posição de um corpo também pode ser alterada como se

fosse um sprite comum, com a vantagem de já termos as colisões com outros corpos processadas pela engine física. Para isso, usamos o método `SetPosition`. Se quisermos deslocar relativamente à posição atual, usamos antes `GetPosition`:

```
function animar() {  
    var posicao = bolaGetPosition();  
    posicao.y -= 1; // Movimento de subida  
    bola.SetPosition(posicao);  
  
    // ...  
}
```

7.3 PROPRIEDADES FÍSICAS

A bola cai no chão, porém não quica. Isso ocorre porque nem ela, nem o chão possuem elasticidade, uma propriedade física que provoca um impulso quando um corpo colide com outro.

As propriedades físicas são configuradas na fixture, ou seja, cada fixture de um corpo pode ter propriedades físicas diferentes. São elas: **elasticidade** (restituição), **atrito** e **densidade**.

Você pode testar a elasticidade no último projeto trabalhado, desde que esteja utilizando o impulso único, disparado em `iniciar`. Deixe no código as linhas indicadas, e comente ou retire qualquer outra mudança de velocidade, ou posição e aplicação de força:

```
function iniciar() {  
    // ...  
  
    var centro = bola.GetWorldCenter();  
    bola.ApplyImpulse(new b2Vec2(3, -10), centro);  
}
```

Na função `criarBola`, defina o atributo `restitution` da `b2FixtureDef`. Este atributo deve receber um valor entre 0 e 1:

```
var fixtureDef = new b2FixtureDef();
fixtureDef.shape = forma;
fixtureDef.restitution = 0.5; // Elasticidade
```

Isso é tudo! Varie os valores: `0.1`, `0.2` etc., até `1.0` e veja a bolinha quicar.

Projeto para demonstrar as propriedades

Vamos criar um projeto semelhante ao anterior, de nome `propriedades`. Neste projeto, faremos uso das três propriedades físicas mencionadas, a começar pelo atrito.

Novamente, no início da seção *Formas*, encontram-se o esqueleto do HTML e o código de inicialização. Copie-os e também as funções `debugDraw` e `animar`. Além disso, copie a função `parede` do projeto `formas`.

Tendo tudo arrumado, vamos criar duas "paredes" no cenário, sendo que uma será o chão e a outra, um muro. Se achar que isto não está muito semântico, sinta-se livre para renomear a função `parede` para algo mais adequado. Aqui, vou manter o nome. Também criaremos um bloco retangular e aplicaremos nele um impulso para a direita:

```
function iniciar() {
    parede(0, 460, 640, 20); // Chão
    parede(620, 0, 20, 460); // Muro
    var bloco = criarBloco();

    ultimoCiclo = Date.now();
    //requestAnimationFrame(animar);
    setTimeout(animar, 1000/60);
```

```
var centro = bloco.GetWorldCenter();
bloco.ApplyImpulse(new b2Vec2(75, 0), centro);
}
```

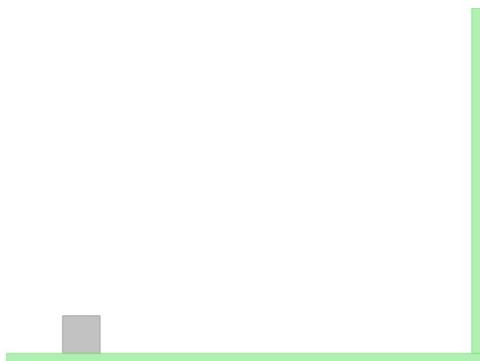


Figura 7.8: Demonstrando atrito, elasticidade e densidade

A fixture do bloco é iniciada com valores considerados médios para as propriedades físicas:

```
function criarBloco() {
    var corpoDef = new b2BodyDef();
    corpoDef.type = b2Body.b2_dynamicBody;
    corpoDef.position.Set(100/escala, 436/escala);

    var forma = new b2PolygonShape();
    forma.SetAsBox(25/escala, 25/escala);

    var fixtureDef = new b2FixtureDef();
    fixtureDef.shape = forma;
    fixtureDef.friction = 0.5;      // Atrito
    fixtureDef.restitution = 0.5;   // Elasticidade
    fixtureDef.density = 1;        // Densidade

    var bloco = mundo.CreateBody(corpoDef);
    bloco.CreateFixture(fixtureDef);
    return bloco;
}
```

Também configure esses valores para a definição de fixture da

parede :

```
function parede(x, y, largura, altura) {  
    // ...  
  
    var fixtureDef = new b2FixtureDef();  
    fixtureDef.shape = forma;  
    fixtureDef.friction = 0.5;      // Atrito  
    fixtureDef.restitution = 0.5;   // Elasticidade  
    fixtureDef.density = 1;        // Densidade  
  
    // ...  
}
```

Nosso laboratório está montado para experiências!

Atrito

Vamos setar valores no atributo `friction`, sabendo que ele pode variar de 0 a 1:

- Coloque o atrito do bloco como 1 e rode a simulação. O bloco nem chega a tocar a parede.
- Coloque os atritos do bloco e da parede como 1. O bloco tomba! É o mesmo atrito provocado caso você aperte com força o freio da frente da bicicleta, em alta velocidade.
- Coloque os atritos como zero. O bloco desliza à vontade, quica e vai embora.

Conclusão: o atrito final depende dos coeficientes de atrito dos dois objetos sendo friccionados.

Elasticidade ou restituição

Vamos zerar os atritos e experimentar com o atributo

`restitution :`

- Coloque a elasticidade do bloco como 1 e veja-o quicar na parede. O bloco chega a saltitar, devido ao impulso que recebe do chão.
- Coloque ambas elasticidades como 1 e veja agora. A parede chuta o bloco para longe!

Conclusão: assim como no atrito, o impulso provocado pela elasticidade também depende das configurações dos dois corpos.

Densidade

A densidade define a **massa** de um corpo. Quanto maior o valor, maior a força ou impulso necessário para deslocá-lo uma dada distância. Experimente mudar o atributo `density` do bloco: `0.5` , `1` , `1.5` , `2` etc. Não há restrição de valor. Apenas observe que, conforme aumentamos a densidade (e, consequentemente, a massa), o deslocamento do corpo com o impulso dado é menor. Você pode obter efeitos semelhantes com densidades diferentes, desde que as forças e impulsos sejam proporcionais.

Experimente variar densidades também na bolinha do projeto `movimento` . Na sua função `criarBola` :

```
var fixtureDef = new b2FixtureDef();
fixtureDef.shape = forma;
fixtureDef.restitution = 0.5;
fixtureDef.density = 1; // Experimente valores diversos
```

Você notará que, quanto menor a densidade, mais alto a bola pula, como acontece com objetos do mundo real. É a diferença entre bater em uma peteca e em uma bigorna.

Quanta teoria! Mas isso é necessário se quisermos dominar a engine. Espero estar tornando as coisas fáceis para você. No próximo capítulo, passaremos a construir a física do jogo de bilhar. Muitos conceitos sobre a engine ainda serão aprendidos.

DOMINANDO O BOX2DWEB

Se você quer realmente dominar a engine, recomendo que invente seus próprios cenários e "personagens", ainda que meros quadradinhos ou bolinhas, dê movimento a eles e experimente com as propriedades físicas. Solte sua imaginação!

CAPÍTULO 8

INICIANDO O JOGO DE BILHAR

Chegou a hora de aplicar o que aprendemos sobre o Box2dWeb em um jogo de verdade. Ao final dessa jornada, você deverá conhecer os aspectos envolvidos na criação de um jogo com uma engine física, e como integrá-la com os conceitos mais básicos, como sprites, loop de animação e interação.



Figura 8.1: Jogo de bilhar a ser construído

No pacote de download, está presente o projeto `jogobilhar`, já pronto e funcional. Recomendo que você o rode em seu celular, a fim de ter uma ideia exata do que será construído. Para dar uma tacada, basta rotacionar o taco com dois dedos

dentro da área circular, puxar o ponteiro da barra de força e tocar o botão POW! no canto superior direito.

Está pronto?

8.1 PREPARAÇÃO DO PROJETO

Siga os seguintes passos para preparar o projeto:

1. Crie um projeto Cordova em seu computador, adicione a plataforma Android e também o plugin de multimídia:

```
cordova create jogo-bilhar
cd jogo-bilhar
cordova platform add android
cordova plugin add org.apache.cordova.media
```

2. Apague todos os arquivos da pasta `www`, porém mantendo as pastas `img` e `js`, para facilitar;
3. Copie a pasta `img` do pacote de download (dentro de `www`) para o seu projeto;
4. Copie os arquivos de som da pasta `sons` para `platforms/android/assets`;
5. Baixe as bibliotecas Hammer.js e Box2dWeb para a pasta `js`, ou copie-as do download;
6. Copie também os arquivos `animacao.js` e `spritesheet.js` do jogo de nave;
7. Configure no `config.xml` o ícone do projeto e a orientação paisagem:

```
<icon src="www/img/icone.png" />
<preference name="Orientation" value="landscape" />
```

O próprio display do aparelho será a mesa, portanto não há a necessidade de virar a imagem constantemente.

Segue o esqueleto inicial do arquivo `index.html`. O script `construtores.js` será criado logo depois. O mundo físico possui **gravidade zero**, exigida por jogos cujo cenário é visto por cima. Para facilitar os trabalhos mais adiante, guardamos a escala em pixels por metro como um atributo do mundo. A classe `Animacao`, criada em meu livro anterior e usada no jogo de nave, vai passar a receber o mundo físico no construtor, pois será responsável por avançar o tempo:

```
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <title>Jogo de Bilhar</title>
    <meta name="viewport" content="width=device-width,
        height=device-height, user-scalable=no,
        initial-scale=1, maximum-scale=1, minimum-scale=1">

    <script src="cordova.js"></script>
    <script src="js/Box2dWeb-2.1.a.3.min.js"></script>
    <script src="js/hammer.min.js"></script>
    <script src="js/construtores.js"></script>
    <script src="js/animacao.js"></script>
    <script src="js/spritesheet.js"></script>

<script>
    var canvas, context, mundo, animacao;
    var ESCALA = 30;

    window.onload = function() {
        canvas = document.getElementById('bilhar');
        context = canvas.getContext('2d');

        // Gravidade zero!
        mundo = new b2World(new b2Vec2(0, 0), true);
        mundo.escala = ESCALA; // Armazenando a escala
```

```

// Passe o mundo para a Animacao
animacao = new Animacao(context, mundo);

debugDraw();
iniciar();
//document.addEventListener('deviceready', iniciar);
}

function iniciar() {
    // Criaremos mais coisas...

    animacao.ligar();
}

function debugDraw() {
    var debugDraw = new b2DebugDraw();
    debugDraw.SetSprite(context);
    debugDraw.SetDrawScale(ESCALA);
    debugDraw.setFillAlpha(0.6);
    debugDraw.SetLineThickness(1.0);
    debugDraw.SetFlags(b2DebugDraw.e_shapeBit);
    mundo.SetDebugDraw(debugDraw);
}
</script>
</head>

<body>
    <canvas id="bilhar" width="700" height="400"></canvas>
</body>

</html>

```

O arquivo construtores.js é linkado logo após o Box2dWeb e iniciará as referências aos principais objetos do Box2dWeb. O único que ainda não foi usado é o b2ContactListener , responsável por tratar eventos de colisão disparados pela engine:

```

var b2Vec2 = Box2D.Common.Math.b2Vec2;
var b2World = Box2D.Dynamics.b2World;
var b2BodyDef = Box2D.Dynamics.b2BodyDef;

```

```
var b2FixtureDef = Box2D.Dynamics.b2FixtureDef;
var b2CircleShape = Box2D.Collision.Shapes.b2CircleShape;
var b2PolygonShape = Box2D.Collision.Shapes.b2PolygonShape;
var b2DebugDraw = Box2D.Dynamics.b2DebugDraw;
var b2Body = Box2D.Dynamics.b2Body;
var b2ContactListener = Box2D.Dynamics.b2ContactListener;
```

Adaptando a classe Animacao

Precisamos adaptar a classe `Animacao` (arquivo `animacao.js`) a uma nova necessidade. Essa classe é responsável por controlar o loop de animação, permitindo colocar e retirar dinamicamente sprites e rotinas de processamento. Ela terá agora de chamar o `Step` do mundo, o método que avança o mundo físico no tempo (seção *Animação do mundo físico*). Por isso, receba-o em seu construtor:

```
function Animacao(context, mundo) {
    this.context = context;
    this.mundo = mundo;

    // ...
}
```

O coração da classe é o método `proximoFrame`, que é nada mais que a função de animação em si. Vamos modificá-lo para que limpe a tela com o fundo verde da mesa de bilhar, avance o tempo do mundo e permita-nos optar entre `setTimeout` e `requestAnimationFrame`. Como o primeiro oferece bom desempenho com o Box2dWeb, podemos nos livrar da preocupação de usar polyfills e suportar aparelhos antigos (que ainda não são tão antigos assim):

```
proximoFrame: function() {
    if ( ! this.ligado ) return;

    // Fundo verde
```

```
var ctx = this.context;
ctx.save();
ctx.fillStyle = '#050';
ctx.fillRect(0, 0, ctx.canvas.width, ctx.canvas.height);
ctx.restore();

var agora = Date.now();
if (this.ultimoCiclo == 0) this.ultimoCiclo = agora;
this.decorrido = agora - this.ultimoCiclo;

// Mundo físico
this.mundo.Step(1/60, 10, 5);
//this.mundo.DrawDebugData(); // Usado em caso de bugs

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

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

for (var i in this.processamentos)
    this.processamentos[i].processar();

this.processarExclusoes();
this.ultimoCiclo = agora;

// Próximo ciclo com setTimeout
var animacao = this;
setTimeout(function() {
    animacao.proximoFrame();
}, 1000/60);
/*
requestAnimationFrame(function() {
    animacao.proximoFrame();
});
*/
},
},
```

IMPORTÂNCIA DO DEBUG DRAW

Perceba que comentei uma chamada a `DrawDebugData` . Se algo der errado no processo, descomente essa linha. Dessa forma, podemos capturar discrepâncias entre a posição real dos corpos e onde os sprites são de fato desenhados. Nem é necessário dizer que precisei disso muitas vezes enquanto criava o jogo.

Abra a página no navegador e vasculhe o Console. Por enquanto, o único erro deverá ser a falta do `cordova.js` . O loop de animação e o mundo físico já estão rodando, embora não haja sprites nem corpos.

8.2 PRIMEIROS SPRITES

Vamos começar com os sprites mais simples: as paredes e as caçapas. O objetivo é mostrar-lhe como integrar o Box2dWeb com a lógica dos sprites.

Paredes da mesa

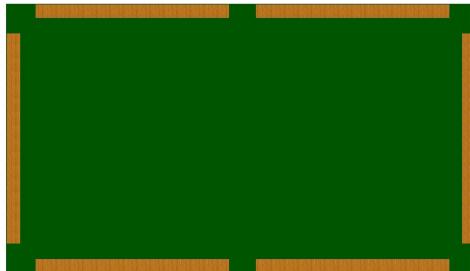


Figura 8.2: Paredes da mesa de bilhar

Na função `iniciar` , coloque uma chamada a `criarParedes` . Como são várias, passamos um array contendo vários objetos (um par de colchetes `[]` contendo chaves `{}`):

```
function iniciar() {
    criarParedes([
        {x: 1, y: 44, largura: 20, altura: 312},
        {x: 679, y: 44, largura: 20, altura: 312},
        {x: 44, y: 1, largura: 288, altura: 20},
        {x: 44, y: 379, largura: 288, altura: 20},
        {x: 372, y: 1, largura: 288, altura: 20},
        {x: 372, y: 379, largura: 288, altura: 20}
    ]);

    animacao.ligar();
}
```

Essa função vai simplesmente percorrer o array e instanciar objetos `Parede` , adicionando-os como sprites ao loop de animação. Perceba que passamos o mundo para o construtor, delegando para a classe a criação do corpo:

```
function criarParedes(arrayDados) {
    var arr = arrayDados;

    for (var i in arr) {
        var p = new Parede(context, arr[i].x, arr[i].y,
            arr[i].largura, arr[i].altura, mundo);
```

```
        animacao.novoSprite(p);
    }
}
```

Crie um link para o arquivo `parede.js`, na pasta `js`:

```
<script src="js/parede.js"></script>
```

E vamos criar esse arquivo. O construtor vai começar criando o corpo no Box2dWeb. Nem precisaríamos configurar um novo `b2BodyDef` como estático, mas fiz isso para dar ênfase. A definição do corpo também pode carregar um atributo `userData`, de uso livre do programador.

Estamos usando-o para associar o corpo ao sprite. Ele será importante durante tratamentos de colisão na engine, para executar ações sobre o sprite. Também, ao definir a posição e o tamanho, usamos o atributo `escala` armazenado no mundo. No mais, sem novidades. Em caso de dúvida, reveja os capítulos *Introdução ao Box2dWeb* e *Configurações e movimento de corpos*:

```
function Parede(context, x, y, largura, altura, mundo) {
    // Corpo físico
    var corpoDef = new b2BodyDef();
    corpoDef.type = b2Body.b2_staticBody;
    corpoDef.userData = this; // Sprite associado

    var fixtureDef = new b2FixtureDef();
    fixtureDef.density = 1;
    fixtureDef.friction = 0.5;
    fixtureDef.restitution = 0.5;

    var meiaLargura = largura / 2;
    var meiaAltura = altura / 2;
    var centroX = x + meiaLargura;
    var centroY = y + meiaAltura;

    // Usando a escala
    corpoDef.position.Set(centroX/mundo.escala,
                          centroY/mundo.escala);
```

```

var retangulo = new b2PolygonShape();
retangulo.SetAsBox(meiaLargura/mundo.escala,
                   meiaAltura/mundo.escala);

fixtureDef.shape = retangulo;

var parede = mundo.CreateBody(corpoDef);
parede.CreateFixture(fixtureDef);

// continua...
}

```

Após criar o corpo, podemos proceder aos afazeres comuns de todo sprite: armazenar seus atributos, atualizá-lo e desenhá-lo. O método `atualizar`, tão importante no jogo de nave, perderá espaço porque quem posiciona os objetos agora é a Box2dWeb.

Recomento que, ao criar seus jogos, mantenha esse método com corpo vazio, para o caso de algum sprite precisar de alguma atualização extra que não tenha a ver com a física. Caso ele se revele desnecessário, ou seja, não haveria nada para executar em lógica de negócio ou de posicionamento, você poderá removê-lo de seus sprites e (não se esqueça!) tirar a chamada feita na classe `Animacao`.

```

function Parede(context, x, y, largura, altura, mundo) {
    // ...

    // Dados do sprite
    this.context = context;
    this.x = x;
    this.y = y;
    this.largura = largura;
    this.altura = altura;
}

Parede.prototype = {
    atualizar: function() {
        //
    }
}

```

```

        },
        desenhar: function() {
            var ctx = this.context;
            ctx.save();
            ctx.fillStyle = ctx.createPattern(IMG_PAREDE, 'repeat');
            ctx.fillRect(this.x, this.y, this.largura, this.altura);
            ctx.restore();
        }
    }
}

```

Podemos carregar a imagem logo ao carregar o script, pois rodando localmente não precisamos ter preocupação com quais imagens já estão carregadas:

```

var IMG_PAREDE = new Image();
IMG_PAREDE.src = 'img/madeira.png';

function Parede(context, x, y, largura, altura, mundo) {
    // ...
}

```

Faça o teste e verifique se o Canvas aparece como na figura *Paredes da mesa de bilhar*.

Caçapas

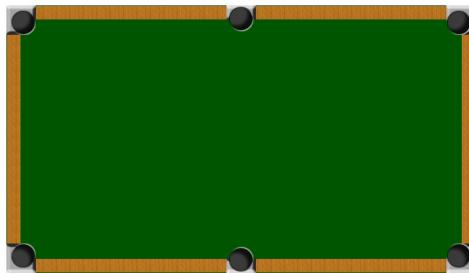


Figura 8.3: Mesa com caçapas

As caçapas da mesa devem ser desenhadas sobre as paredes,

portanto, os sprites devem ser adicionados à animação depois. A spritesheet das caçapas é formada por seis imagens e, para facilitar, vamos criá-las na mesma sequência em que estão desenhadas no arquivo. Além das imagens, cada caçapa terá um pequeno corpo a ser usado como sensor de colisão:

```
function iniciar() {
    // ...

    criarCacapas([
        {imgX: 0, imgY: 0, sensorX: 30, sensorY: 30},
        {imgX: 328, imgY: 0, sensorX: 350, sensorY: 25},
        {imgX: 656, imgY: 0, sensorX: 670, sensorY: 30},
        {imgX: 0, imgY: 356, sensorX: 30, sensorY: 370},
        {imgX: 328, imgY: 356, sensorX: 350, sensorY: 375},
        {imgX: 656, imgY: 356, sensorX: 670, sensorY: 370}
    ]);

    animacao.ligar();
}
```



Figura 8.4: Spritesheet das caçapas

Cada caçapa criada usará uma coluna da spritesheet, atribuída pelo número de ordem (variável de loop `i`):

```
function criarCacapas(arrayDados) {
    var arr = arrayDados;

    for (var i in arr) {
        var c = new Cacapa(context, arr[i].imgX, arr[i].imgY,
                           arr[i].sensorX, arr[i].sensorY, mundo);
        c.spritesheet.coluna = i;
        animacao.novoSprite(c);
    }
}
```

Não se esqueça de linkar o arquivo da classe:

```
<script src="js/cacapa.js"></script>
```

E vamos criar a classe `Cacapa`. Os corpos físicos serão pequenos quadrados, os sensores, posicionados estratégicamente nos pontos de entrada de cada caçapa. O atributo `isSensor` da definição da fixture faz com que ela não impeça a passagem de outro corpo, enquanto a engine pode reportar que eles estão em contato. As bolas atravessarão os sensores como se não houvesse um corpo ali.

```
var SENSOR = 1; // Meia largura/altura

function Cacapa(context, imgX, imgY, sensorX, sensorY, mundo) {
    // Corpo do Box2dWeb (sensor)
    var corpoDef = new b2BodyDef();
    corpoDef.position.Set(sensorX/mundo.escala,
                          sensorY/mundo.escala);
    corpoDef.type = b2Body.b2_staticBody;
    corpoDef.userData = this;

    var quadrado = new b2PolygonShape();
    quadrado.SetAsBox(SENSOR/mundo.escala, SENSOR/mundo.escala);

    var fixtureDef = new b2FixtureDef();
    fixtureDef.shape = quadrado;
    fixtureDef.isSensor = true; // Sensor!

    this.corpo = mundo.CreateBody(corpoDef);
    this.corpo.CreateFixture(fixtureDef);

    // continua ...
}
```

Da mesma forma que com as paredes, vamos carregar a imagem e armazenar os atributos do sprite:

```
var SENSOR = 1;

var SPRITE_CACAPA = new Image();
```

```

SPRITE_CACAPA.src = 'img/cacapa-spritesheet.png';

function Cacapa(context, imgX, imgY, sensorX, sensorY, mundo) {
    // ...

    // Atributos do sprite
    this.context = context;
    this.mundo = mundo;
    this.imgX = imgX;
    this.imgY = imgY;
    this.sensorX = sensorX;
    this.sensorY = sensorY;
    this.spritesheet = new Spritesheet(context, SPRITE_CACAPA,
        1, 6);
}

}

```

E também desenhar a caçapa. No método `desenhar` do sprite, chamamos o método homônimo da `spritesheet`, que desenha conforme a coluna atribuída anteriormente:

```

Cacapa.prototype = {
    atualizar: function() {
        //
    },
    desenhar: function() {
        this.spritesheet.desenhar(this.imgX, this.imgY);
    }
}

```

Faça o teste e veja se as caçapas aparecem. Para vermos os sensores, abra a classe `Animacao`, descomente a chamada ao `DrawDebugData` e comente o loop que chama `desenhar` nos sprites:

```

// Descomente esta linha
this.mundo.DrawDebugData();

// ...

// Comente estas
//for (var i in this.sprites)
//    this.sprites[i].desenhar();

```



Figura 8.5: Debug Draw da mesa de bilhar

Sempre faça este procedimento para certificar-se de que o mundo físico está de acordo com o que você deseja em seus jogos.

8.3 AS BOLAS DE BILHAR

As bolas do jogo representam um tipo de sprite com mais detalhes para lidar. Em primeiro lugar, é inútil armazenarmos atributos de posição, pois a engine física os deslocará o tempo inteiro. A posição de desenho de cada bola deverá sempre ser lida da engine. Faremos uso de uma folha de sprites animada e, não bastasse isso, o ângulo de desenho da bola deverá corresponder ao ângulo da sua trajetória. Cuidaremos disto nesta e nas próximas seções. Preparado?

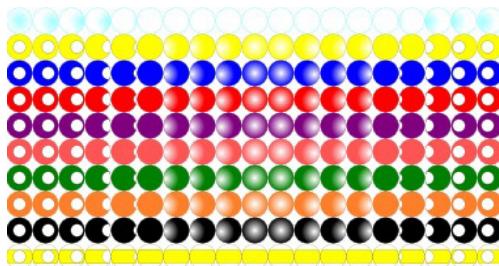


Figura 8.6: Não sou nenhum artista, portanto perdoem-me

Vamos inicialmente declarar variáveis que farão referências às bolas:

```
var canvas, context, mundo, animacao;  
var ESCALA = 30;  
var bolaBranca, bolasColoridas;
```

E mandar criá-las após as caçapas. O número de cada bola corresponde à linha da sua cor na spritesheet (figura anterior). A bola branca se encontra na linha zero.

```
function iniciar() {  
    // ...  
  
    bolaBranca = criarBolaBranca(150, 200); // x, y  
    bolasColoridas = criarBolasColoridas([  
        {x: 519, y: 200, numero: 1},  
        {x: 600, y: 200, numero: 2},  
        {x: 580, y: 190, numero: 3},  
        {x: 580, y: 210, numero: 4},  
        {x: 540, y: 190, numero: 5},  
        {x: 540, y: 210, numero: 6},  
        {x: 560, y: 180, numero: 7},  
        {x: 560, y: 220, numero: 8},  
        {x: 560, y: 200, numero: 9}  
    ]);  
  
    animacao.ligar();  
}
```

As funções `criarBolaBranca` e `criarBolasColoridas` instanciam objetos `Bola`, configuram o atributo `id` com o número e as adicionam como sprites na animação. Para a bola branca, é melhor criar uma constante (`BOLA_BRANCA`) do que escrever diretamente um valor (zero, negativo ou o que quer que seja), para definir que se trata da bola branca:

```
function criarBolaBranca(x, y) {  
    var b = new Bola(context, x, y, 12, mundo); // raio = 12  
    b.id = BOLA_BRANCA;
```

```

animacao.novoSprite(b);
return b;
}

function criarBolasColoridas(arrayDados) {
    var arr = arrayDados;
    var bolas = new Array(arr.length);

    for (var i in arr) {
        // raio = 10
        var b = new Bola(context, arr[i].x, arr[i].y, 10, mundo);
        b.id = arr[i].numero;
        animacao.novoSprite(b);
        bolas[i] = b;
    }

    return bolas;
}

```

Temos de criar a classe `Bola`. Faça o link para o script:

```
<script src="js/bola.js"></script>
```

E segue a primeira versão. Definimos a constante `BOLA_BRANCA`, carregamos a spritesheet, criamos o corpo e guardamos os atributos essenciais. A definição de corpo possui um atributo `bullet`, que aprimora a detecção de colisão em alta velocidade (mais detalhes a seguir). Não é necessário guardar atributos de posição no sprite:

```

var BOLA_BRANCA = 0;

var BOLA_SPRITE = new Image();
BOLA_SPRITE.src = 'img/bola-spritesheet.png';

function Bola(context, x, y, raio, mundo) {
    // Corpo do Box2dWeb
    var corpoDef = new b2BodyDef();
    corpoDef.position.Set(x/mundo.escala, y/mundo.escala);
    corpoDef.type = b2Body.b2_dynamicBody;
    corpoDef.bullet = true; // Colisão de corpos velozes
    corpoDef.userData = this;
}

```

```
var fixtureDef = new b2FixtureDef();
fixtureDef.shape = new b2CircleShape(raio/mundo.escala);
fixtureDef.density = 1;
fixtureDef.friction = 0.5;
fixtureDef.restitution = 0.7;

this.corpo = mundo.CreateBody(corpoDef);
this.corpo.CreateFixture(fixtureDef);

// Atributos da bola
this.id = 0;
this.context = context;
this.raio = raio;
this.mundo = mundo;

// Spritesheet
var sheet = new Spritesheet(context, BOLA_SPRITE, 10, 19);
this.spritesheet = sheet;
}
```

ATRIBUTO BULLET

Por padrão, o Box2D e ports calculam a posição de um corpo no tempo e o posicionam diretamente, da mesma forma como estamos acostumados a fazer com nossos sprites sem engine física. No entanto, corpos muito pequenos e velozes podem passar por barreiras "de uma vez" entre um quadro e outro, impedindo a detecção da colisão.

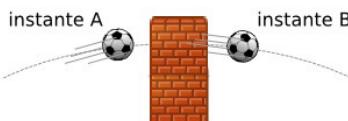


Figura 8.7: Sem um detector de colisão apurado, esta bola atravessaria o muro

Quando uma definição de corpo possui o valor `true` no atributo `bullet`, o Box2D realiza uma detecção mais apurada, considerando os pontos intermediários da trajetória. Isto não é feito por padrão porque requer muito processamento; você deve configurar como `bullet`s apenas corpos muito velozes que apresentarem o problema de atravessar outros inadvertidamente.

Para desenhar, façamos um primeiro teste. Primeiro lemos a posição da bola com o método `GetPosition` do corpo. Depois convertemos as coordenadas para pixels, multiplicando-as pela escala. Por último, desenhamos círculos básicos para nos certificarmos de que as bolinhas estão sendo desenhadas nas

posições corretas:

```
Bola.prototype = {
    atualizar: function() {
        },

    desenhar: function() {
        // Posição
        var posicao = this.corpo.GetPosition();
        var x = posicao.x * this.mundo.escala;
        var y = posicao.y * this.mundo.escala;

        // Simples por enquanto
        var ctx = this.context;
        ctx.beginPath();
        ctx.arc(x, y, this.raio, 0, Math.PI*2);
        ctx.fill();
    }
}
```

Rode a página e veja se bolas pretas aparecem sobre a mesa. Vamos então aplicar a spritesheet. A linha corresponde ao atributo `id` da bola. O método `proximoQuadro`, para quem não conhece do meu outro livro, avança a animação no tempo. Em seguida, mandamos desenhar o quadro atual:

```
desenhar: function() {
    var posicao = this.corpo.GetPosition();
    var x = posicao.x * this.mundo.escala;
    var y = posicao.y * this.mundo.escala;

    var sheet = this.spritesheet;
    sheet.linha = this.id;
    sheet.proximoQuadro();
    sheet.desenhar(x - this.raio, y - this.raio,
                   this.raio * 2, this.raio * 2);
}
```

As bolinhas ficam girando mesmo paradas, o que será tratado mais à frente. Primeiro, quero chamar a atenção para algo errado:

o método `desenhar` da classe `Spritesheet` não recebe a largura e a altura! Precisamos disso, pois a bola branca possui um tamanho diferente das outras, determinado pelo raio.

Vamos modificar esse método para que receba opcionalmente a largura e a altura do desenho. As chamadas antigas que não passarem estes parâmetros continuarão desenhando o tamanho real do quadro da spritesheet:

```
desenhar: function(x, y, largDestino, altDestino) {
    var largOrigem = this.imagem.width / this.numColunas;
    var altOrigem = this.imagem.height / this.numLinhas;

    if (!largDestino) largDestino = largOrigem;
    if (!altDestino) altDestino = altOrigem;

    this.context.drawImage(
        this.imagem,
        largOrigem * this.coluna,
        altOrigem * this.linha,
        largOrigem,
        altOrigem,
        x,
        y,
        largDestino,
        altDestino
    );
}
```

Perceba que agora a spritesheet respeita as dimensões passadas, e a bola branca aparece maior. Faça o teste com o Debug Draw também.

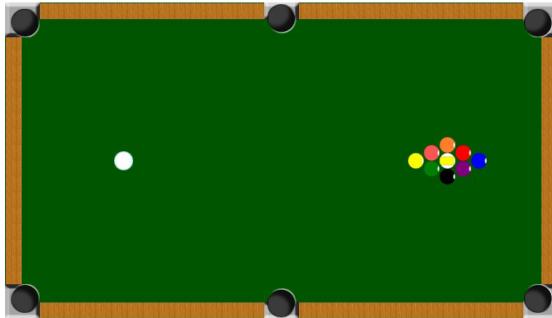


Figura 8.8: Bolas posicionadas na mesa

8.4 EXECUTANDO A TACADA

Está pensando no que eu estou pensando? Que tal aplicar um impulso na bola branca e simular uma tacada? Para tratar o desenho correto das bolas se movimentando, precisamos provocar primeiro um movimento. Em `iniciar`, acrescente uma chamada à função `tacada` :

```
function iniciar() {  
    // ...  
    tacada();  
}
```

Nessa função, começamos definindo o ângulo e a força que, lembre-se, são atributos presentes na classe `Taco` criada no capítulo *Interações avançadas com Hammer.js*. Criamos um objeto `Tacada`, que é acrescentado como um processamento do loop de animação, e chamamos um método `disparar` :

```
function tacada() {  
    // Virão do Taco depois :D  
    var angulo = 0;  
    var forca = 100;  
  
    var t = new Tacada(bolaBranca, bolasColoridas);
```

```
animacao.novoProcessamento(t);
t.disparar(angulo, forca);
}
```

Temos de criar a classe `Tacada`. Faça um novo link:

```
<script src="js/tacada.js"></script>
```

E segue um esqueleto para a classe. Preste atenção ao atributo `rodando`: qualquer processamento pode ser ligado e desligado testando-se um atributo desse tipo, permitindo-nos controlar melhor sua configuração e até mesmo setar atributos importantes antes de começar o processamento:

```
function Tacada(branca, coloridas) {
    this.branca = branca;
    this.coloridas = coloridas;
    this.rodando = false;
}

Tacada.prototype = {
    disparar: function(angulo, forca) {

    },
    processar: function() {
        if (!this.rodando) return;

        // Faremos algo aqui
    }
}
```

No método `disparar`, precisamos determinar os componentes x e y da força a ser aplicada, com a ajuda do seno e do cosseno (ver box adiante). Depois, aplicamos essa força como um impulso na bola branca, sinalizamos que o processamento está rodando e marcamos o instante do início, para cronometrar o tempo:

```
disparar: function(angulo, forca) {
```

```
var radianos = angulo * Math.PI / 180;
var forcaX = forca * Math.cos(radianos);
var forcaY = forca * Math.sin(radianos);

var forca = new b2Vec2(forcaX, forcaY);
var centro = this.branca.corpo.getWorldCenter();
this.branca.corpo.ApplyImpulse(forca, centro);

this.rodando = true;
this.inicio = Date.now();
},
```

DECOMPOSIÇÃO DE VETOR EM X E Y

Dados o valor de uma força, também chamado *módulo* dessa força, e o ângulo de aplicação, podemos determinar seus componentes *x* e *y* pelas fórmulas:

$$x = \text{força} * \cos(\text{ângulo})$$
$$y = \text{força} * \sin(\text{ângulo})$$

A tacada já acontece, ainda com as bolinhas girando loucamente. As bolas que entram na caçapa seguem adiante para fora, pois atravessam os corpos sensores. O mundo físico não possui barreiras.

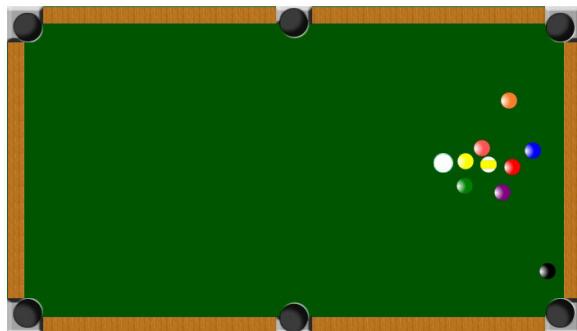


Figura 8.9: Pow!

Só que a ação não termina! Os corpos continuam em movimento porque as paredes não oferecem resistência suficiente para dissipar sua energia. No método `processar`, vamos verificar o tempo decorrido e mandar as bolas frearem gradativamente. Por isso, guardamos todas elas em um array único, usando o método `concat` do array `coloridas`, para juntá-las com a branca. Depois, o método `IsAwake` do corpo verifica se ele está acordado, pois configuramos o mundo do Box2dWeb para adormecer corpos que pararem. A variável `mexendo` sinalizará se ainda há ou não bolas em movimento:

```
processar: function() {
    if (!this.rodando) return;

    var decorrido = Date.now() - this.inicio;
    var mexendo = false;
    var bolas = this.coloridas.concat([this.branca]);

    for (var i in bolas) {
        var b = bolas[i];

        // Monitorar o movimento
        if (b.corpo.IsAwake()) {
            mexendo = true;
```

```

    // Frear a bola
    if (decorrido >= 7000)
        b.frear(5.0);
    else if (decorrido >= 5000)
        b.frear(1.0);
    else if (decorrido >= 3000)
        b.frear(0.7);
}
}

// continua...
}

```

Os valores passados para o método `frear` da `Bola`, a ser criado logo mais, representarão a desaceleração que os corpos sofrerão. Esses valores, e também os intervalos de tempo, foram obtidos após vários testes, de acordo com o tempo que considerei adequado para a animação durar — não há mágica para determinar isso, você pode fazer seus próprios testes depois que esta parte estiver pronta.

Após iterar por todas as bolas, se nenhuma mais estiver se mexendo, devemos reverter a desaceleração de todas elas, setando o valor zero. Isso é imprescindível para que elas não parem e passem a ir em sentido contrário! Sinalizamos o fim da execução da tacada e chamamos um callback por onde o jogo poderá dar continuidade a seu processamento:

```

processar: function() {
    // ...

    if ( ! mexendo ) {
        // Reverter a desaceleração
        for (var i in bolas)
            if (!bolas[i].foraDeJogo) bolas[i].frear(0);

        if (this.apostacada) this.apostacada();
        this.rodando = false;
        console.log('tacada finalizada');
    }
}

```

```
    }  
}
```

É boa prática "documentar" o atributo `aposTacada` inicializando-o como `null` no construtor:

```
this.aposTacada = null;
```

Precisamos agora criar o método `frear` na classe `Bola`. Os métodos `SetLinearDamping` e `SetAngularDamping` configuram uma desaceleração gradativa para as velocidades linear (de trajetória) e angular (de rotação), respectivamente:

```
frear: function(desaceleracao) {  
    this.corpo.SetLinearDamping(desaceleracao);  
    this.corpo.SetAngularDamping(desaceleracao);  
}
```

Pode experimentar a tacada! Quando a animação completar 7 segundos, a desaceleração vai ser tal que elas pararão em pouco tempo e a mensagem "tacada finalizada" poderá ser vista no console. Experimente configurar outros ângulos e forças na função `tacada` !

8.5 ÂNGULO DA TRAJETÓRIA

Vamos resolver o problema da animação da spritesheet. Se fosse apenas animar quando estiverem em movimento, seria fácil. Mas o desenho da bola deve ser **rotacionado** de acordo com a sua trajetória:



Figura 8.10: Bolas rotacionadas em trajetórias diferentes

Para nossa sorte, dado um vetor com componentes (x , y), como a velocidade, o JavaScript nos fornece o método `Math.atan2`, que devolve o ângulo desse vetor com relação à horizontal, exigindo somente os parâmetros na ordem invertida: (y , x). Obtemos a velocidade do corpo com `GetLinearVelocity` e usamos `Math.atan2` para determinar o ângulo. Vamos reconstruir o método `desenhar` da classe `Bola`:

```
desenhar: function() {
    // Ângulo da trajetória
    var velocidade = this.corpo.GetLinearVelocity();
    var vX = velocidade.x;
    var vY = velocidade.y;
    var angulo = Math.atan2(vY, vX); // y, x

    // continua ...
},
```

Agora, peço que se detenha e preste bastante atenção. A classe `Spritesheet` possui o atributo `intervalo`, que define o tempo entre um quadro e outro. Esse intervalo é **inversamente proporcional** à velocidade: quanto mais rápida a bola, menor o intervalo da animação, fazendo-a girar mais rápido.

Mas há um pequeno porém: uma velocidade **negativa** é matematicamente menor do que uma positiva de 0,001 ou mesmo zero. Mas o sinal negativo aqui significa apenas o **sentido** em que a bola percorre a mesa. Devemos trabalhar em valores absolutos,

considerando que -20 é maior que 5, por exemplo.

Tendo os componentes `vX` e `vY` da velocidade, vamos obter seus valores absolutos e ver qual o maior. Se for zero, não definimos intervalo (atribuímos `null`). Se for maior que zero, o intervalo será o inverso da velocidade ($1/\text{velocidade}$). Isso, claro, gera intervalos em valores decimais menores que 1, portanto multiplicamos por algum fator, no caso 50, para obter intervalos maiores.

Quanto tudo estiver pronto, você poderá fazer vários testes alterando esse valor e ver que o intervalo entre os quadros da spritesheet aumenta ou diminui, mas sempre de forma inversa à velocidade do movimento:

```
desenhar: function() {  
    // ...  
  
    // Velocidade da animação  
    var velMax = Math.max( Math.abs(vX), Math.abs(vY) );  
    var intervalo = ( velMax == 0 ? null : 1 / velMax * 50 );  
  
    // continua ...  
},
```

Confesso, esta foi a parte que mais me deu trabalho de bolar!

Finalmente, temos de obter a posição da bola e desenhá-la rotacionada. Como aprendemos na seção *Rotacionando um taco de bilhar*, para desenhar um objeto rotacionado, precisamos transladar o Canvas para o local desejado para depois rotacionar:

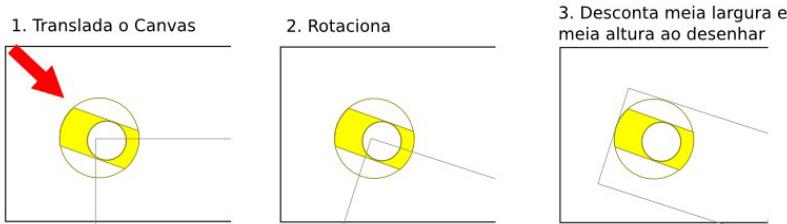


Figura 8.11: Como rotacionar a bola no Canvas

Esse conceito é muito importante para desenharmos qualquer objeto rotacionado! Leia então a posição do corpo e faça as alterações no contexto do Canvas:

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

    // Posição
    var posicao = this.corpoGetPosition();
    var x = posicao.x * this.mundo.escala;
    var y = posicao.y * this.mundo.escala;

    // DESENHAR ROTACIONADA
    var ctx = this.context;

    ctx.save();

    // Ponto de desenho
    ctx.translate(x, y);

    // Rotaciona
    ctx.rotate(angulo);

    // continua ...
},
```

Podemos agora fazer o desenho, descontando o raio da bola da posição atual do Canvas. Caso tenhamos definido um intervalo anteriormente (bola em movimento), nós o aplicamos na spritesheet e mandamos avançar a animação. Não se esqueça de

chamar o `restore` quando fizer alterações drásticas no contexto gráfico!

```
desenhar: function() {
    // ...
    // Desenha
    var sheet = this.spritesheet;
    sheet.linha = this.id;

    if (intervalo) {
        sheet.intervalo = intervalo;
        sheet.proximoQuadro();
    }

    sheet.desenhar(-this.raio, -this.raio, this.raio * 2,
                    this.raio * 2);

    ctx.restore();
},
```

Terminamos a primeira etapa! Temos todos os sprites necessários e já somos capazes de conduzir uma tacada no jogo, disparando-a e sabendo quando a ação terminou.

No próximo capítulo, incorporaremos ao jogo as classes `Taco` e `BarraForca`, criadas no capítulo *Interações avançadas com Hammer.js*, para adicionar interação, e aprenderemos como reagir a colisões no Box2dWeb, disparando sons e fazendo as bolas sumirem quando colidirem com os sensores da caçapa. Espero que esteja gostando!

CAPÍTULO 9

INTERAÇÃO COM O JOGADOR E LÓGICAS DE NEGÓCIO

Vamos agora permitir aos jogadores de nosso bilhar virtual controlar a sua tacada e encaçapar as bolas, fazendo-as desaparecer.

No capítulo *Interações avançadas com Hammer.js*, nós criamos o projeto `taco-bilhar`, que contém os controles de interação com o jogo deste livro. No caso, um controle de rotação do taco e uma barra de regulagem da força da tacada. Copie para o projeto atual (`jogo-bilhar`) os arquivos `taco.js` e `barra-força.js`, e também a biblioteca `Hammer.js`. Você pode obter esses arquivos no pacote de download ou da pasta onde você salvou seus exercícios.

Certifique-se de que o projeto já possui todos os sons e imagens copiados para as pastas corretas. Em caso de dúvidas, reveja o início do capítulo *Iniciando o jogo de bilhar*.

Confira o conjunto dos scripts que devem ser linkados até aqui:

```
<script src="cordova.js"></script>
<script src="js/Box2dWeb-2.1.a.3.min.js"></script>
```

```
<script src="js/hammer.min.js"></script>
<script src="js/construtores.js"></script>
<script src="js/spritesheet.js"></script>
<script src="js/animacao.js"></script>
<script src="js/bola.js"></script>
<script src="js/parede.js"></script>
<script src="js/cacapa.js"></script>
<script src="js/tacada.js"></script>
<script src="js/taco.js"></script>
<script src="js/barra-forca.js"></script>
```

Vamos começar!

9.1 CONTROLES DA TACADA

Preparação

Na pasta das imagens, existe o arquivo `tacada.png`, que representa o botão que dispara a tacada. Você pode usar outra figura, se desejar. Na seção `<body>`, acrescente a tag da imagem logo após a tag `<canvas>`:

```
<body>
  <canvas id="bilhar" width="700" height="400"></canvas>
  
</body>
```

E vamos posicionar este botão via CSS, também aproveitando para deixar o Canvas responsivo, como aprendemos na seção *Toque em Canvas responsivo*. Na seção `<head>`, acrescente uma tag `<style>` e faça as configurações a seguir:

```
body {
  margin: 0;
  width: 100%;
  height: 100%;
  background: #333;
}
```

```

canvas {
    height: 100%;
    width: auto;
    display: block;
    margin: 0 auto;
}

#tacada {
    position: absolute;
    right: 20px;
    top: 20px;
}

```

Acrescente também referências ao Hammer.js, aos controles e ao arquivo de som da tacada, nas declarações de variáveis da página:

```

var canvas, context, mundo, animacao;
var ESCALA = 30;
var bolaBranca, bolasColoridas;
var hammer, taco, barraForca, somTacada;

```

E modifique o evento `onload` da página para carregar o som e o Hammer.js. Vamos, por enquanto, deixar comentadas as linhas que só rodam em dispositivo móvel, para podermos realizar testes no console do Google Chrome no desktop:

```

window.onload = function() {
    canvas = document.getElementById('bilhar');
    context = canvas.getContext('2d');

    //somTacada = new Media('file:///android_asset/colisao.mp3');

    mundo = new b2World(new b2Vec2(0, 0), true);
    mundo.escala = ESCALA;

    animacao = new Animacao(context, mundo);
    hammer = new Hammer(canvas);

    debugDraw();
    iniciar();
}

```

```
//document.addEventListener('deviceready', iniciar);
}
```

Em `iniciar`, fizemos uma chamada à função `tacada` somente para testar a lógica. Retire essa chamada:

```
function iniciar() {
    // ...
    // Remova
    //tacada();
}
```

Criação dos controles

As classes `Taco` e `BarraForca` estão recebendo as imagens pelo construtor, seguindo a linha de carregar as imagens na inicialização. Esta é uma boa prática para jogos que devem ser publicados na internet, mas para um jogo local vamos simplificar os construtores. Abra o arquivo `taco.js`, insira as linhas que carregam a imagem e modifique o construtor como segue:

```
var IMG_TACO = new Image();
IMG_TACO.src = 'img/taco.png';

function Taco(context, hammer) {
    this.context = context;
    this.imagem = IMG_TACO;

    // ...
}
```

O mesmo em `barra-forca.js`:

```
var IMG_PONTEIRO = new Image();
IMG_PONTEIRO.src = 'img/ponteiro.png';

function BarraForca(context, taco, hammer) {
    this.context = context;
    this.ponteiro = IMG_PONTEIRO;
```

```
// ...
}
```

Temos então de criar os controles. Em `iniciar`, instanciaremos o `Taco` e configuraremos o disparo pelo toque no botão da tacada. Ainda manteremos comentado o comando que rebobina o som:

```
function iniciar() {
    // Paredes, caçapa e bolas
    // ...

    taco = new Taco(context, hammer);
    taco.raio = 120;
    taco.x = 125;
    taco.y = canvas.height / 2 - 50;

    var imgTacada = document.getElementById('tacada');
    imgTacada.addEventListener('touchstart', function() {
        //somTacada.seekTo(0);
        taco.darTacada(50);
    });

    taco.aposTacada = tacada;
    animacao.novoSprite(taco);

    // Aqui virá a barra de força
    animacao.ligar();
}
```

O mesmo se sucederá com a barra de força:

```
function iniciar() {
    // ...

    barraForca = new BarraForca(context, taco, hammer);
    barraForca.x = taco.x - 100;
    barraForca.y = taco.y + taco.raio + 25;
    barraForca.largura = 200;
    barraForca.altura = 50;
    animacao.novoSprite(barraForca);
```

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

Como os controles estão sendo adicionados como sprites na animação, estes devem ter o método `atualizar`, ainda que vazio. No capítulo anterior, eu disse que, se o `atualizar` se mostrar desnecessário em algum jogo, você poderá removê-lo dos sprites e também sua chamada pela classe `Animacao`. Modifique ambos os controles:

```
Taco.prototype = {
  atualizar: function() {
    // ...
  },
  // ...
}

BarraForca.prototype = {
  atualizar: function() {
    // ...
  },
  // ...
}
```

Faça o teste no navegador e veja se não há erros de codificação no console, exceto o do `cordova.js`. Você pode também rodar no seu celular e testar os controles.

9.2 EXECUTANDO A TACADA NO ÂNGULO CORRETO

A função `tacada`, atualmente, está definindo arbitrariamente a força e o ângulo. Vamos modificá-la para que leia esses dados do controle do taco. Também vamos impedir uma tacada com força

nula e mandar esconder os controles enquanto a ação acontece. Perceba que criamos um callback, `bolasPararam`, para reagir quando a animação tiver terminado:

```
function tacada() {
    if (taco.forca == 0) return;

    // Esconder os controles
    taco.podeDesenhar = false;
    barraForca.podeDesenhar = false;

    // Realizar a tacada
    var tacada = new Tacada(bolaBranca, bolasColoridas);
    animacao.novoProcessamento(tacada);
    tacada.aposTacada = bolasPararam;
    tacada.disparar(taco.rotacao, taco.forca);

    //somTacada.play();
}

function bolasPararam() {
    // Aqui executaremos outras regras de negócio!

    taco.podeDesenhar = true;
    barraForca.podeDesenhar = true;
}
```

Nos construtores, tanto do `Taco` quanto da `BarraForca`, inicie o novo atributo `podeDesenhar` como `true`:

```
this.podeDesenhar = true;
```

Em seus métodos `desenhar`, faça-os considerarem o atributo `logo` no início:

```
desenhar: function() {
    if (!this.podeDesenhar) return;

    // ...
}
```

Rode no celular. Gire o taco e aperte o botão da tacada. Deve

funcionar, mas a bola não parece ir na direção correta!

Na verdade, ela vai. Se você deixar o ângulo padrão (0°), a bola branca é impulsionada para a direita, porém o taco é desenhado em pé, porque sua imagem é em pé. Isso é fácil de ser resolvido, basta desenhar a imagem 90° ($\pi/2$) a mais que a rotação definida no controle. No método `desenhar do Taco`, modifique a linha que calcula o ângulo da imagem em radianos:

```
// Taco  
var radianos = this.rotacao * Math.PI / 180 + Math.PI / 2;
```

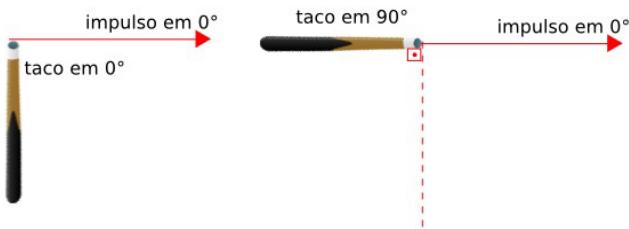


Figura 9.1: Somando 90° à imagem do taco, ele fica alinhado corretamente ao ângulo da tacada

Experimente novamente e veja que a bola branca vai na direção para onde você girou o taco.

TESTANDO A TACADA PELO CONSOLE

Pelo celular é muito mais interativo, mas demora para carregar. Testes rápidos podem ser feitos pelo Google Chrome. A barra de força pode ser puxada com o mouse, pois é gerida pelo Hammer.js. A rotação exige dois dedos na tela touch, mas podemos setar o atributo e disparar uma tacada diretamente pelo console:

```
taco.rotacao = 60; // em graus  
tacada();  
  
    > taco.rotacao = 60;  
    60  
    > tacada()  
    undefined  
    > |
```

Figura 9.2: Dando comandos ao jogo pelo console do Google Chrome

9.3 POSICIONANDO O TACO

Embora tudo esteja começando a funcionar bem, ainda está estranho ver o taco sempre no mesmo local, enquanto a bola branca vaga pela mesa. Vamos criar uma nova função que cuidará de posicionar a imagem do taco junto à bola branca, no início e após cada tacada. Na função `iniciar`, comece chamando `posicionarTaco` após a sua instanciação:

```
taco = new Taco(context, hammer);  
taco.raio = 120;
```

```
taco.x = 125;  
taco.y = canvas.height / 2 - 50;  
  
// Desenho do taco a partir da bola branca  
posicionarTaco();
```

A função `posicionarTaco` vai ler a posição da bola branca no mundo físico e configurar os novos atributos `brancaX` e `brancaY`, a serem usados somente para o desenho do taco. Os atributos `x` e `y` não mudam, pois continuarão correspondendo ao centro do controle circular. Também é preciso especificar um `raioDistancia` para afastar a ponta do taco do centro da bola, evitando que os desenhos se sobreponham:

```
function posicionarTaco() {  
    var pos = bolaBranca.corpoGetPosition();  
    var x = pos.x * ESCALA;  
    var y = pos.y * ESCALA;  
  
    taco.brancaX = x;  
    taco.brancaY = y;  
    taco.raioDistancia = bolaBranca.raio;  
}
```

No construtor do `Taco`, inicie esses atributos para documentá-los devidamente:

```
this.brancaX = 0;  
this.brancaY = 0;  
this.raioDistancia = 0;
```

E no método `desenhar`, o `translate` passa a ser feito para esse ponto desejado. O deslocamento do taco não será apenas de acordo com a força, mas também somado com o raio de distância do centro da bola branca:

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

```

// Taco
var radianos = this.rotacao * Math.PI / 180 + Math.PI / 2;
ctx.save();
ctx.translate(this.brancaX, this.brancaY);
ctx.rotate(radianos);
ctx.drawImage(this.imagem, -7,
              this.deslocamento() + this.raioDistancia,
              this.imagem.width, this.imagem.height);
ctx.restore();
},

```

Sabemos quando a tacada terminou, portanto nesse momento vamos reposicionar o taco e zerar as forças. Acrescente as seguintes linhas em `bolasPararam` :

```

function bolasPararam() {
    // ...

    posicionarTaco();
    barraForca.forca = 0;
    taco.forca = 0;
}

```

Façamos um teste mais amplo no celular. Ative o evento `deviceready` , descomente os comandos envolvendo o `somTacada` e certifique-se de que a tacada dispara o som e o taco acompanha a bola branca a cada jogada. Isto está ficando divertido!

9.4 TRATAMENTO DAS COLISÕES

Só a tacada vai fazer som? Precisamos fazer os choques entre bolas, entre uma bola e uma parede, e entre uma bola e um sensor de caçapa produzirem sons também. Portanto, vamos aprender como reagir a colisões entre corpos no Box2dWeb. A *detecção de colisões* entre diversas formas geométricas já é realizada automaticamente pela engine, bastando-nos realizar o seu

tratamento no jogo.

No evento `onload` da página, mande carregar todos os sons:

```
somTacada = new Media('file:///android_asset/colisao.mp3');
somMadeira = new Media('file:///android_asset/knock.mp3');
somCacapa = new Media('file:///android_asset/hit.mp3');
```

Em `iniciar`, vamos instanciar um novo objeto, `ColisaoListener`, a ser criado logo mais. Este objeto será o responsável por receber os eventos de colisão do Box2dWeb. Para inseri-lo no mundo, usamos `SetContactListener`:

```
function iniciar() {
    // ...

    colisaoListener = new ColisaoListener(somTacada, somMadeira,
                                         somCacapa);
    mundo.SetContactListener(colisaoListener);

    animacao.ligar();
}
```

Atualize as variáveis da página como segue:

```
var canvas, context, mundo, animacao;
var ESCALA = 30;
var bolaBranca, bolasColoridas;
var hammer, taco, barraForca;
var somTacada, somMadeira, somCacapa, colisaoListener;
```

E faça o link para `colisao-listener.js`:

```
<script src="js/colisao-listener.js"></script>
```

Vamos agora criar a nova classe `ColisaoListener`. O construtor somente armazenará referências aos sons:

```
function ColisaoListener(somBola, somMadeira, somCacapa) {
    this.somBola = somBola;
    this.somMadeira = somMadeira;
```

```
    this.somCacapa = somCacapa;  
}
```

Um objeto ouvinte de colisões do Box2dWeb deve seguir uma interface, ou seja, implementar um certo conjunto de métodos. No entanto, como nem todos eles podem nos interessar, existe o objeto `b2ContactListener`, que já implementa todos eles, porém com corpos vazios. Devemos fazer nossa classe herdar os métodos existentes em `b2ContactListener` e sobrescrever aqueles que nos interessarem. Em JavaScript, uma herança entre classes pode ser simulada definindo um objeto como protótipo de outro. Na sequência do código:

```
// ColisaoListener herda os métodos de b2ContactListener  
ColisaoListener.prototype = new b2ContactListener();
```

Continuando, vamos redefinir o método `BeginContact`. Desta vez não podemos atribuir um novo protótipo, mas modificar o método do protótipo recém-criado. Esse método é chamado pelo Box2dWeb logo no início de um contato entre fixtures de corpos, e recebe um parâmetro com os dados desse contato.

```
ColisaoListener.prototype.BeginContact = function(contato) {  
    // tratamento da colisão aqui  
}
```

Preste atenção ao "caminho" até chegar aos corpos e seus sprites associados. Os métodos `GetFixtureA` e `GetFixtureB`, do objeto `contato`, obtêm as fixtures em colisão. Para saber quais seus corpos correspondentes, chamamos `GetBody`. E o sprite tinha sido associado ao corpo como um *user data*, portanto `GetUserData` deve nos retornar uma `Bola`, um `Taco` ou uma `Parede` !

```
ColisaoListener.prototype.BeginContact = function(contato) {  
    var corpoA = contato.GetFixtureA().GetBody();
```

```

var corpoB = contato.GetFixtureB().GetBody();

var objetoA = corpoA.GetUserData();
var objetoB = corpoB.GetUserData();

// continua...
}

```

Testemos os tipos dos objetos. Para facilitar, salvamos os testes em variáveis. Temos de testar a possibilidade de serem duas bolas se chocando, perguntando se `objetoA` e `objetoB` são do tipo `Bola`, e a possibilidade de uma `Bola` se chocar com `Parede` ou `Cacapa`, não importando qual é *A* ou *B*:

```

ColisaoListener.prototype.BeginContact = function(contato) {
    // ...

    var bola1 = (objetoA instanceof Bola ? objetoA : null);
    var bola2 = (objetoB instanceof Bola ? objetoB : null);
    var bola = bola1 || bola2; // Quem quer que seja a Bola!

    var parede = (objetoA instanceof Parede ? objetoA :
                  objetoB instanceof Parede ? objetoB : null);

    var cacapa = (objetoA instanceof Cacapa ? objetoA :
                  objetoB instanceof Cacapa ? objetoB : null);

    // continua...
}

```

Podemos, enfim, reagir às colisões, disparando os respectivos sons. No caso de ser bola com caçapa, chamamos o callback `encacapou` na bola, de modo a executar a lógica de negócio:

```

ColisaoListener.prototype.BeginContact = function(contato) {
    // ...

    if (bola1 && bola2) {
        this.somBola.play();
        this.somBola.seekTo(0); // Rebobinar para a próxima
    }
}

```

```

else if (bola && parede) {
    this.somMadeira.play();
    this.somMadeira.seekTo(0);
}
else if (bola && cacapa) {
    this.somCacapa.play();
    this.somCacapa.seekTo(0);
    if (bola.encacapou) bola.encacapou(bola);
}
}

```

No construtor da `Bola`, documente esse callback e também crie um atributo `foraDeJogo`, com o valor `false`, para podermos executar alguns testes adiante:

```

this.encacapou = null;
this.foraDeJogo = false;

```

No `index.html`, em `criarBolaBranca`, atribua um callback antes do `return`:

```

function criarBolaBranca(x, y) {
    // ...
    b.encacapou = encacapouBranca;
    return b;
}

```

E atribua outro em `criarBolasColoridas`, dentro do loop que cria as bolas:

```

function criarBolasColoridas(arrayDados) {
    // ...

    for (var i in arr) {
        // ...
        b.encacapou = encacapouColorida;
    }

    // ...
}

```

Precisamos criar esses callbacks. O da bola branca vai

repositioná-la, e o das coloridas vai excluí-las quando encaçapadas. Mas aqui é preciso muita cautela. *Não devemos mudar o estado de um corpo durante um tratamento de colisão, pois o passo (Step) ainda está sendo processado!* Em outras palavras, nada de reposicionar ou excluir os corpos: simplesmente não funcionará. E agora?

Temos de agendar essa tarefa para após o Step . Felizmente, já temos um mecanismo para isso: podemos adicionar processamentos ao loop de animação, para serem executados ao fim de cada ciclo!

No callback encacapouBranca , crie um processamento. Lá dentro, sinalize que a bola está fora do jogo, pare seu movimento e posicione-a do lado de fora da mesa, para não atrapalhar o movimento das outras. No final, o processamento deve ser excluído:

```
function encacapouBranca(bola) {  
    // Somente ao fim do ciclo  
    animacao.novoProcessamento({  
        processar: function() {  
            bola.foraDeJogo = true;  
            bola.corpo.SetLinearVelocity(new b2Vec2(0, 0));  
            bola.corpo.SetPosition(new b2Vec2(-1000, -1000));  
  
            // Excluir para executar apenas uma vez  
            animacao.excluirProcessamento(this);  
        }  
    });  
}
```

Em bolasPararam , que responde ao fim de uma tacada, vamos verificar se a bola branca foi encaçapada, reposicioná-la e retorná-la para o jogo. Isso deve ser feito antes da chamada a posicionarTaco :

```

function bolasPararam() {
    if (bolaBranca.foraDeJogo) {
        bolaBranca.corpo.SetPosition(
            new b2Vec2(150/ESCALA, 200/ESCALA));
        bolaBranca.foraDeJogo = false;
    }

    // posicionarTaco() por aqui...
}

```

E temos o outro callback, `encacapouColorida`. Quando uma bola colorida é encaçapada, ela deve sair do jogo. Essa lógica de negócio será delegada para a classe `Bola`:

```

function encacapouColorida(bola) {
    animacao.novoProcessamento({
        processar: function() {
            bola.tirarDoJogo();
            animacao.excluirProcessamento(this);
        }
    });
}

```

O método `tirarDoJogo`, na classe `Bola`, executará todas as ações necessárias: excluir o corpo do mundo físico, excluir o sprite da animação e sinalizar que aquela bola está fora do jogo:

```

tirarDoJogo: function() {
    this.mundo.DestroyBody(this.corpo);
    this.animacao.excluirSprite(this);
    this.foraDeJogo = true;
}

```

De volta à classe `Tacada`, verifique se a bola *não* (símbolo `!`) está fora de jogo, além de estar acordada, para considerá-la ativa. Dessa forma, qualquer bola que for embora pelas caçapas não precisa aguardar um bom tempo sendo freada:

```

processar: function() {
    // ...

```

```
// Monitorar o movimento  
if (!b.foraDeJogo && b.corpo.IsAwake()) {  
  
    // ...  
}
```

A interação com o jogador está finalizada! Fomos capazes de:

- integrar os controles do taco e da barra de força ao jogo;
- alinhar o desenho do taco com o ângulo de tacada e colocar sua ponta na borda da bola;
- disparar uma tacada no ângulo e com a força definidos nos controles;
- aguardar as bolas pararem para tomar outras medidas;
- "sumir" com a bola branca e retorná-la ao início do jogo, se for encaçapada;
- excluir de verdade as bolas coloridas encaçapadas.

No próximo capítulo, veremos como o computador poderá calcular uma tacada automática, aproveitando-se do fato de que a Box2dWeb pode fazer as simulações físicas somente em memória. Brinque um pouco, o jogo começa agora a ficar viciante!

CAPÍTULO 10

SIMULAÇÕES INTERNAS

Neste capítulo, discuto como podemos fazer a máquina pensar para determinar sua jogada. Usando uma engine física, calcular a jogada automática não requer conhecimentos em inteligência artificial. Sabendo que o Box2D e suas variações em outras linguagens (*ports*) trabalham em memória, podemos realizar diferentes tacadas por baixo dos panos, analisar os resultados e mostrar na tela aquela que for mais vantajosa.

Isso parece simples, e realmente é, mas há considerações importantes a se fazer:

- a cada simulação feita em memória, devemos retornar todas as bolas para as posições em que se encontravam no início;
- o processamento da tacada só termina após passado um período de tempo, quando se começa a frear o movimento das bolas;
- nosso tratador de colisões está disparando sons e tirando as bolas do jogo.

Vamos pensar em cada um destes problemas por vez.

10.1 REALIZANDO VÁRIAS SIMULAÇÕES A

PARTIR DO INÍCIO

Simularemos diversas tacadas, em diversos ângulos. Só para testar, forçaremos o computador a dar uma tacada logo ao fim da função `iniciar`:

```
function iniciar() {  
    // ...  
    tacadaComputador();  
}
```

Na função `tacadaComputador`, criamos um objeto `TacadaAutomatica` e o mandamos fazer os testes, devolvendo os dados da tacada que deu os melhores resultados. Tudo de forma abstraída, delegando para essa classe o serviço pesado:

```
function tacadaComputador() {  
    var automatica = new TacadaAutomatica(mundo, bolaBranca,  
                                            bolasColoridas);  
    var melhorTacada = automatica.realizarTestes();  
    taco.rotacao = melhorTacada.rotacao;  
    taco.forca = melhorTacada.forca;  
    tacada();  
}
```

Crie o arquivo `tacada-automatica.js`. Para simular uma tacada, temos de distinguir a bola branca das coloridas (veja a classe `Tacada`), e também precisamos do mundo para simular os Step's:

```
function TacadaAutomatica(mundo, branca, coloridas) {  
    this.mundo = mundo;  
    this.branca = branca;  
    this.coloridas = coloridas;  
}  
  
TacadaAutomatica.prototype = {  
    realizarTestes: function() {  
    }  
}
```

}

Em `realizarTestes`, vamos delegar para a classe `Snapshot` a tarefa de salvar e recuperar a posição das bolas. Chamamos `salvar` no início e `reverter` após cada simulação. A simulação é feita no método `executar`, que recebe o ângulo e a força a serem testados, e deverá retornar os dados que obteremos da tacada:

```
realizarTestes: function() {
  var estadoJogo = new Snapshot(this.mundo);
  estadoJogo.salvar(); // Guarda no início

  var resultados = [];

  // De 5 em 5 graus para não onerar o processador
  for (var i = 0; i <= 360; i += 5) {
    var tacada = this.executar(i, 75); // ângulo, força
    resultados.push(tacada);
    estadoJogo.reverter(); // Reverte após cada uma
  }

  // Como decidir qual foi a melhor simulação?
  return resultados[0];
}, // vírgula pois terá o executar
```

O salto de 5 em 5 graus é mais um detalhe a que não cheguei logo de primeira. Fiz vários testes ao fim da implementação, os quais recomendo você fazer também. Dar 360 tacadas, voltando as bolas para suas posições após cada uma, leva bastante tempo. Também evitei testar diferentes valores de força para cada ângulo. Se quiser, você pode, como um exercício, encontrar um balanceamento adequado entre quantos ângulos e quantas forças podem ser testadas para cada um.

Ainda não temos como decidir qual simulação deu o melhor resultado, pois não definimos quais regras de jogo vamos adotar.

Faremos isso no próximo capítulo. Mas vamos pelo menos nos certificar de que as simulações estão acontecendo e que o desempenho desse processamento seja aceitável.

No método `executar`, criamos um objeto `Tacada` e "empurramos" o processamento com um loop `while` em vez de usar `requestAnimationFrame` ou `setTimeout`. O loop é controlado pela variável `continua`, que é setada para `false` quando o processamento reportar o fim da ação no callback `aposTacada`:

```
executar: function(angulo, forca) {
    var tacada = new Tacada(this.branca, this.coloridas);

    var continua = true;
    tacada.aposTacada = function() {
        continua = false;
    }

    tacada.disparar(angulo, forca);

    while (continua) {
        this.mundo.Step(1/60, 10, 5);
        tacada.processar();
    }

    // Obteremos depois outros dados
    return { rotacao: angulo, forca: forca };
}
```

Nossa prática até aqui tem sido avançar o mundo físico, em cada ciclo do loop de animação, no mesmo tempo medido pelo relógio do computador, criando uma simulação em tempo real. Ainda assim, nada impede que todos os passos (`Step s`) de uma tacada sejam simulados de forma seguida, em loops `for` ou `while` comuns, sem nos atrelarmos ao relógio. A Box2dWeb calcula a posição dos corpos no tempo, independentemente de esse

tempo ter se passado no relógio ou não.

10.2 SALVANDO E RECUPERANDO O ESTADO DO MUNDO FÍSICO

Para tudo isso funcionar, precisamos implementar a classe `Snapshot`. Nela, guardaremos as posições das bolas. Crie o arquivo `snapshot.js`:

```
function Snapshot(mundo) {
    this.mundo = mundo;
    this.posicoes = [];
}
Snapshot.prototype = {
    salvar: function() {
        },
    reverter: function() {
        }
}
```

O método `salvar` vai fazer loop nos corpos e guardar as posições. Perceba que não guardamos diretamente os objetos devolvidos por `GetPosition`, e, sim, copiamos os valores para novos objetos. Fazemos isso porque variáveis de objetos são **referências**, ou seja, estaríamos na verdade referenciando os objetos de posição reais dos corpos, que poderão estar alterados na hora de reverter.

```
salvar: function() {
    var arrPosicoes = [];
    var corpo = this.mundo.GetBodyList();

    while (corpo) {
        var pos = corpoGetPosition();
        var obj = { corpo: corpo, x: pos.x, y: pos.y };
        arrPosicoes.push(obj);
    }
}
```

```
        corpo = corpo.GetNext();
    }

    this.posicoes = arrPosicoes;
},
```

ALGORITMO PARA ITERAR POR TODOS OS CORPOS

A primeira referência é obtida por `GetBodyList` ; as próximas, por `GetNext` . Quando for devolvido `null` , os corpos se acabaram:

```
var corpo = this.mundo.GetBodyList();

while (corpo) {
    // ...

    corpo = corpo.GetNext();
}
```

O método `reverter` só precisa fazer um loop no array de posições guardadas e chamar `SetPosition` para reatribuí-las aos corpos:

```
reverter: function() {
    for (var i in this.posicoes) {
        var pos = this.posicoes[i];
        pos.corpo.SetPosition(new b2Vec2(pos.x, pos.y));
    }
}
```

Quero frisar que, para a nossa necessidade aqui, apenas guardar e retornar as posições dos corpos é suficiente, já que cada jogada é feita com as bolas paradas. Você pode estender a classe `Snapshot` para guardar velocidades lineares e angulares, com os métodos `GetLinearVelocity` e `GetAngularVelocity` ,

respectivamente, e recuperá-las com os `Set`s correspondentes.

Não se esqueça de linkar os scripts das duas classes novas:

```
<script src="js/tacada-automatica.js"></script>
<script src="js/snapshot.js"></script>
```

10.3 EXECUTANDO A TACADA EM TEMPO CURTO

Se executarmos o jogo no ponto em que está, veremos a tela parada enquanto as simulações acontecem. Dá até para ouvir os sons. Mas lembre-se de que a `Tacada` faz uma desaceleração progressiva das bolas, sendo que depois de 7 segundos ela "pisa mais fundo" no freio. Fazendo isso 72 vezes ($360/5$), imagine o tempo que leva. Não adianta nada fazermos os `Step`s de forma seguida se a classe `Tacada` só trabalha com o relógio.



Figura 10.1: Aguardando as simulações acontecerem

Vamos flexibilizá-la para que trabalhe também com contagem de quadros. Sabendo que cada quadro leva $1/60$ de segundo, basta contar os quadros para saber quanto tempo uma tacada levou, sem precisar esperar esse tempo passar de verdade. Podemos assim

fazer simulações mais rápidas, enquanto a animação pode (e deve) continuar dependendo do relógio, acontecendo na velocidade em que os eventos físicos aconteceriam no mundo real.

No método `executar` da `TacadaAutomatica`, estamos instanciando a `Tacada`. Configure dois novos atributos, `porRelogio` e `tempoAvanco`:

```
executar: function(angulo, forca) {
    var tacada = new Tacada(this.branca, this.coloridas);
    tacada.porRelogio = false;
    tacada.tempoAvanco = 1/60;

    // ...
}
```

No construtor da `Tacada`, inicie o atributo `porRelogio` como `true`, para não quebrar o comportamento padrão. O `tempoAvanco` pode ser iniciado como zero, para documentação de atributos, pois só será usado se for para simulação rápida:

```
function Tacada(branca, coloridas) {
    // ...
    this.porRelogio = true;
    this.tempoAvanco = 0;
}
```

Na simulação rápida, devemos fazer uma contagem de quadros para determinar o tempo avançado. Em `disparar`, inicie um atributo `quadro` com zero:

```
disparar: function(angulo, forca) {
    // ...
    this.quadro = 0;
},
```

E em `processar`, incrementamos o contador e decidimos se é por relógio ou por contagem de quadros. Se for por relógio,

determinamos o tempo decorrido da mesma forma como estávamos fazendo. Se for por contagem, sabemos quantos quadros foram avançados e quanto tempo foi simulado, pelo atributo `tempoAvanco` :

```
processar: function() {
    if (!this.rodando) return;
    this.quadro++;

    // Quantos segundos já se passaram
    var sete, cinco, tres;
    if (this.porRelogio) {
        // Esta linha vem para cá
        var decorrido = Date.now() - this.inicio;
        sete = (decorrido >= 7000);
        cinco = (decorrido >= 5000);
        tres = (decorrido >= 3000);
    }
    else {
        var segundos = this.quadro * this.tempoAvanco;
        sete = (segundos >= 7);
        cinco = (segundos >= 5);
        tres = (segundos >= 3);
    }

    // ...
}
```

Basta-nos então usar as variáveis dos testes para frear as bolas. Modifique o trecho de `processar` em que isso ocorre:

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

    // Frear a bola
    if (sete)
        b.frear(5.0);
    else if (cinco)
        b.frear(1.0);
    else if (tres)
        b.frear(0.7);
```

```
// ...  
}
```

A classe `Tacada` poderia trabalhar somente com contagem de quadros, mesmo no loop de animação, pois este já controla o tempo entre os passos. Mas isso nos obrigaria, a cada tacada criada, a informar o `tempoAvanco` a ser simulado, razão por que preferi mantê-la flexível. Esteja à vontade para experimentar essa abordagem, se quiser.

Experimente executar no dispositivo móvel neste momento. O tempo de simulação já melhorou bastante, embora ainda esteja longe do adequado, e também escutamos os sons disparados pelo tratador de colisão. As bolas que foram encaçapadas nesse percurso somem, pois é o que mandamos fazer nesses casos. Tendo resolvido primeiro os problemas de desempenho, vamos resolver estes.

10.4 TRATADOR DE COLISÕES PARA A SIMULAÇÃO

Vamos criar outra classe tratadora de colisão voltada para as simulações. Esta classe não disparará sons e detectará inicialmente somente as encaçapadas.

Na função `tacadaComputador`, altere o tratador de colisão no início, e retorne ao original após o fim das simulações, mas antes de realizar a tacada "real" do jogo:

```
function tacadaComputador() {
```

```
    mundo.SetContactListener(simulacaoListener);

    // ...

    mundo.SetContactListener(colisaoListener);
    tacada();
}

}
```

Declare a variável `simulacaoListener` na página:

```
var canvas, context, mundo, animacao;
var ESCALA = 30;
var bolaBranca, bolasColoridas;
var hammer, taco, barraForca, somTacada;
var somTacada, somMadeira, somCacapa;
var colisaoListener, simulacaoListener;
```

Instancie o seu objeto em `iniciar` (pode ser junto com o primeiro):

```
function iniciar() {
    // ...

    simulacaoListener = new SimulacaoListener();
    colisaoListener = new ColisaoListener(somTacada, somMadeira,
                                         somCacapa);
    mundo.SetContactListener(colisaoListener);

    // ...
}
```

Não se esqueça de linkar o arquivo de script:

```
<script src="js/simulacao-listener.js"></script>
```

E vamos criá-lo. Perceba que é uma versão bastante reduzida do primeiro, já que sua função por enquanto é reportar somente contato entre Bola e Cacapa :

```
function SimulacaoListener() {
    this.encacapou = null;
}
```

```

SimulacaoListener.prototype = new b2ContactListener();

SimulacaoListener.prototype.BeginContact = function(contato) {
    var objA = contato.GetFixtureA().GetBody().GetUserData();
    var objB = contato.GetFixtureB().GetBody().GetUserData();

    var bola = (objA instanceof Bola ? objA :
                objB instanceof Bola ? objB : null);
    var cacapa = (objA instanceof Cacapa ? objA :
                  objB instanceof Cacapa ? objB : null);

    if (bola && cacapa && this.encacapou)
        this.encacapou(bola);
}

```

Respondendo ao callback

Faça com que a `TacadaAutomatica` receba esse simulador no construtor. Como essa classe é responsável por calcular a jogada do computador, ela fornecerá um callback que verificará as bolas encaçapadas:

```

function tacadaComputador() {
    mundo.SetContactListener(simulacaoListener);
    var automatica = new TacadaAutomatica(mundo, bolaBranca,
                                           bolasColoridas, simulacaoListener);
    // ...
}

```

No seu construtor, recebemos o colisor e criamos um callback que chama o novo método `aoEncacapar`:

```

function TacadaAutomatica(mundo, branca, coloridas, colisor) {
    // ...

    var tacadaAuto = this;
    colisor.encacapou = function(bola) {
        tacadaAuto.aoEncacapar(bola);
    }
}

```

```
}
```

Nesse método, precisaríamos fazer algo para tirar a bola da mesa, para evitar que continue colidindo com outras (caso não tenha saído para fora). Porém, lembre-se de que, durante o tratamento de colisão, isso não pode ser feito (seção *Tratamento das colisões*, capítulo anterior). Logo, aqui vamos apenas registrar que a bola foi encaçapada:

```
// Chamado dentro do Step pelo tratador de colisão
aoEncacapar: function(bola) {
    this.encacapadas.push(bola);
}
```

Em executar , inicie o array encacapadas antes de processar todos os passos. Após cada Step , podemos chamar um método para cuidar das bolas que deverão "sumir", deslocarEncacapadas . Por fim, tendo as bolas encaçapadas devidamente registradas pelo callback, podemos retorná-las como resultado:

```
executar: function(angulo, forca) {
    // ...

    // Zerar o array antes de processar os ciclos
    this.encacapadas = [];

    while (continua) {
        this.mundo.Step(0.1, 10, 5);
        tacada.processar();

        // Após o Step podemos dar um jeito nas bolas que se foram
        this.deslocarEncacapadas();
    }

    // Ao fim podemos retornar os resultados (return no final!!)
    return {
        rotacao: angulo,
        forca: forca,
```

```
    encacapadas: this.encacapadas  
  };  
},
```

Em `deslocarEncacapadas`, podemos fazer com as bolas o mesmo que fazemos com a bola branca em uma tacada "normal": parar seu movimento e deslocá-las para além das paredes, para não interferir no movimento das que ficam:

```
deslocarEncacapadas: function() {  
  var arr = this.encacapadas;  
  
  for (var i in arr) {  
    arr[i].corpo.SetLinearVelocity(new b2Vec2(0, 0));  
    arr[i].corpo.SetPosition(new b2Vec2(-1000, -1000));  
  }  
}
```

Se você rodar o aplicativo, o aparelho já consegue "pensar" por uns instantes, testando tacadas em todos os ângulos que determinamos, antes de disparar uma tacada para valer. Ainda falta escolher qual o melhor ângulo, mas para isso temos de ter claras as regras do jogo.

No próximo capítulo discutiremos mais a fundo que regras vamos adotar, pois existem muitas variedades de bilhar e sinuca. Poderemos então facilmente determinar a melhor tacada, bem como gerenciar a partida, alternando jogadas do humano e do computador e determinando quando um venceu. Falta pouco agora!

CAPÍTULO 11

FINALIZANDO O JOGO

Estamos quase lá! Precisamos agora determinar as regras do jogo para podermos fazer a máquina raciocinar sobre suas tacadas e sobre a partida. Por que deixei isto somente para o final do livro, se o correto é iniciarmos o desenvolvimento com as regras definidas? *Porque existem inúmeras variações de jogos de bilhar ou sinuca.*

Tendo programado toda a mecânica de movimento, podemos reaproveitá-la em diferentes versões do jogo. Se quiser, mantenha uma cópia reservada do jogo até este ponto. Você pode depois implementar as regras de outra variação que costuma jogar.



Figura 11.1: Resultado final a ser atingido

11.1 REGRAS DO BOLA 9

Aqui implementaremos o *Bola 9*, uma variedade de bilhar, de acordo com o que está descrito em <http://pt.wikipedia.org/wiki/Bilhar>:

- o objetivo do jogo é encaçapar a bola de número 9;
- a bola branca deve sempre atingir primeiro a de menor número ainda na mesa;
- se encaçapar alguma bola colorida, desde que a partir da bola de ordem, o jogador da vez continua;
- se a bola 9 for encaçapada dentro dessas condições, em qualquer momento, o jogador da vez ganha a partida;
- se a branca não acertar nenhuma bola ou acertar uma bola inválida, o jogador passa a vez, mesmo que encaçape;
- se a branca for encaçapada, ela retorna para sua posição inicial. O mesmo ocorre com a bola 9 encaçapada de maneira ilegítima. Nesses casos, o jogador também passa a vez.

Primeiro, vamos declarar todas as variáveis de página responsáveis pelo tratamento das regras. Confira a relação final de declarações:

```
var canvas, context, mundo, animacao;
var ESCALA = 30;
var bolaBranca, bolasColoridas;
var hammer, taco, barraForca;
var somTacada, somMadeira, somCacapa;
var colisaoListener, simulacaoListener;
var bolaOrdem, jogadorDaVez, primeiraAtingida, encacapadas,
mostrador;
var JOG_HUMANO = 1, JOG_COMPUTADOR = 2;
```

Agora, configure o início de uma partida. Na função

`iniciar`, retire a chamada direta a `tacadaComputador` e mude para definir quem é jogador da vez e qual a primeira bola a ser acertada:

```
function iniciar() {  
    // ...  
  
    // O final fica:  
    jogadorDaVez = JOG_HUMANO;  
    bolaOrdem = bolasColoridas[0];  
    animacao.ligar();  
  
    // Remova  
    //tacadaComputador();  
}
```

SORTEANDO O JOGADOR

Se desejar, use `Math.random` para determinar quem sai na primeira partida.

Determinando a primeira bola atingida

Antes de tudo, vamos determinar qual foi a primeira bola atingida pela bola branca em uma tacada. Para isso, temos de ouvir eventos de colisão entre bolas. Isso já está sendo feito em `ColisaoListener`, mas não em `SimulacaoListener`. Altere a classe para que detecte essa situação e chame um callback:

```
SimulacaoListener.prototype.BeginContact = function(contato) {  
    var objA = contato.GetFixtureA().GetBody().GetUserData();  
    var objB = contato.GetFixtureB().GetBody().GetUserData();  
  
    // Detecte ambas as bolas  
    var bola1 = (objA instanceof Bola ? objA : null);
```

```

var bola2 = (objB instanceof Bola ? objB : null);
var bola = (bola1 || bola2);

var cacapa = (objA instanceof Cacapa ? objA :
               objB instanceof Cacapa ? objB : null);

if (bola && cacapa && this.encacapou)
    this.encacapou(bola);

// Callback de colisão
if (bola1 && bola2)
    this.colisaoBolas(bola1, bola2);
}

```

Em `TacadaAutomatica`, no método `executar`, inicie o atributo `primeiraAtingida` como `null` antes do loop que processa os passos da tacada de simulação:

```

this.encacapadas = [];
this.primeiraAtingida = null;

while (continua) {
    // ...
}

```

No construtor, crie um callback que verifica qual a primeira bola atingida pela branca. Se o atributo já estiver setado, nada deve ser feito. Se a branca não pegar em nenhuma, o evento não ocorrerá e o atributo continuará `null`:

```

function TacadaAutomatica(mundo, branca, coloridas, colisor) {
    // ...

colisor.colisaoBolas = function (bola1, bola2) {
    if (tacadaAuto.primeiraAtingida) return;

    if (bola1.id == BOLA_BRANCA)
        tacadaAuto.primeiraAtingida = bola2;
    else if (bola2.id == BOLA_BRANCA)
        tacadaAuto.primeiraAtingida = bola1;
}
}

```

Analogamente, a classe `ColisaoListener` também deve disparar um callback em caso de colisão entre bolas:

```
if (bola1 && bola2) {  
    this.somBola.play();  
    this.somBola.seekTo(0);  
    if (this.colisaoBolas) this.colisaoBolas(bola1, bola2);  
}
```

Claro, documente o callback no construtor:

```
function ColisaoListener(somBola, somMadeira, somCacapa) {  
    // ...  
    this.colisaoBolas = null;  
}
```

Agora, atribua esse callback na página HTML, na função `iniciar` (pode ser logo após a instanciação):

```
colisaoListener = new ColisaoListener(somTacada, somMadeira,  
                                      somCacapa);  
colisaoListener.colisaoBolas = colisaoEntreBolas;
```

Por fim, crie a função `colisaoEntreBolas`, com a mesma lógica de determinação da primeira bola atingida, mas referenciando a variável `primeiraAtingida` da página:

```
function colisaoEntreBolas(bola1, bola2) {  
    if (primeiraAtingida) return;  
  
    if (bola1.id == BOLA_BRANCA)  
        primeiraAtingida = bola2;  
    else if (bola2.id == BOLA_BRANCA)  
        primeiraAtingida = bola1;  
}
```

Analizando as jogadas

Sabendo qual a primeira bola que foi atingida pela branca em uma tacada, seja em simulação ou numa tacada real do jogo,

podemos aplicar as regras para analisar a jogada. Vamos delegar essa tarefa para uma nova classe, `AnaliseJogada`, para podermos usar as mesmas regras em todas as situações. Portanto, faça seu link na página:

```
<script src="js/analise-jogada.js"></script>
```

Antes de programá-la, é preciso definir o que esperar dela, conforme nossa prática habitual de primeiro "mandar" na classe, para depois fazê-la obedecer. Na classe `TacadaAutomatica`, no método `realizarTestes`, vamos fazer a análise ao fim de cada ciclo do loop que testa os ângulos, guardando os resultados. Tão logo ocorrer algo favorável, podemos parar os testes e retornar o objeto com os dados, economizando tempo de processamento. Perceba que, para analisarmos corretamente uma tacada, temos de saber qual a bola de ordem (da vez), por isso a recebemos como parâmetro no início:

```
// Receba a bola de ordem
realizarTestes: function(bolaOrdem) {
    // ...
    for (var i = 0; i <= 360; i += 5) {
        // ...
        var analise = new AnaliseJogada();
        analise.analisar(this.primeiraAtingida, bolaOrdem,
                          tacada.encacapadas);
        // Guarda a análise junto no resultado
        tacada.analise = analise;
        // Por ordem de prioridade
        if (analise.ganhouJogo)
            return tacada;
        else if (analise.encacapou)
            return tacada;
    }
}
```

```
// ...
}
```

Ao fim do método, se não foi encontrada nenhuma tacada favorável, ficamos com a primeira que for legal:

```
realizarTestes: function(bolaOrdem) {
    // ...

    // Encontrar a primeira tacada válida
    for (var i in resultados) {
        if (resultados[i].analise.jogadaLegal)
            return resultados[i];
    }

    // Se não foi possível fazer nenhuma válida...
    return resultados[0];
},
```

Na função `tacadaComputador` , passe a bola de ordem, já definida anteriormente, quando chamar `realizarTestes` :

```
function tacadaComputador() {
    // ...
    var melhorTacada = automatica.realizarTestes(bolaOrdem);
    // ...
}
```

Agora comece a implementar a classe `AnaliseJogada` . No construtor, documentaremos todas as situações possíveis:

```
function AnaliseJogada() {
    this.jogadaLegal = false;
    this.encacapouBranca = false;
    this.encacapou = false;
    this.ganhouJogo = false;
}
```

No método `analisar` , começamos montando um array somente com os números das bolas, a fim de facilitar a análise.

Definimos que uma jogada é legal caso a primeira bola atingida seja a bola de ordem e a branca não tenha sido encaçapada. Em caso de jogada legal, podemos determinar se houve encaçapada normal (para o jogador poder continuar) e se houve vitória (bola 9 encaçapada em uma jogada legal):

```
AnaliseJogada.prototype = {
    analisar: function(primeiraAtingida, bolaOrdem, encacapadas){
        // Números (IDs)
        var ids = [];

        for (var i in encacapadas)
            ids.push(encacapadas[i].id);

        // Jogada foi válida?
        this.encacapouBranca = (ids.indexOf(BOLA_BRANCA) >= 0);
        this.jogadaLegal = (primeiraAtingida &&
                            primeiraAtingida.id == bolaOrdem.id &&
                            !this.encacapouBranca);

        // Resultados
        this.encacapou = (this.jogadaLegal && ids.length > 0);
        this.ganhouJogo =
            (this.jogadaLegal && ids.indexOf(9) >= 0);
    }
}
```

O aparelho já levará bem menos tempo para encontrar uma tacada que dê algum resultado ou, pelo menos, seja válida pelas regras do jogo. Se quiser fazer o teste, coloque novamente uma chamada a `tacadaComputador` ao fim da função `iniciar`.

Controlando a partida

Analogamente, precisamos fazer as mesmas verificações em uma tacada real, seja do computador, seja do jogador, para controlar o andamento da partida. Na função `tacada`, vamos iniciar as variáveis `encacapadas` e `primeiraAtingida`, da

mesma forma que na simulação:

```
function tacada() {  
    // ...  
  
    // Realizar a tacada  
    encacapadas = [];  
    primeiraAtingida = null;  
    var tacada = new Tacada(bolaBranca, bolasColoridas);  
  
    // ...  
}
```

Nas funções `encacapouBranca` e `encacapouColorida`, acrescente a bola encaçapada no array. Não é preciso fazer nenhuma distinção, pois a classe `AnaliseJogada` cuida de verificar se a bola branca foi encaçapada, requerendo apenas um array com todas elas:

```
function encacapouBranca(bola) {  
    encacapadas.push(bola);  
  
    // ...  
}  
  
function encacapouColorida(bola) {  
    encacapadas.push(bola);  
  
    // ...  
}
```

O callback `bolasPararam` é chamado pela `Tacada` quando a animação finaliza. Vamos chamar ali uma nova função responsável por analisar a jogada e determinar o rumo da partida:

```
function bolasPararam() {  
    // ...  
  
    analisarJogada();  
}
```

Chegamos ao coração da partida, ao árbitro do jogo. A função analisarJogada solicita os serviços da classe AnaliseJogada da mesma forma que a TacadaAutomatica . Se houve vitória, é exibida uma mensagem e a página é recarregada. Se encaçapou, não alteramos nada para permitir ao jogador da vez continuar. Porém, caso não haja encaçapada legítima, trocamos o jogador da vez:

```
function analisarJogada() {
    var analise = new AnaliseJogada();
    analise.analisar(primeiraAtingida, bolaOrdem, encacapadas);

    // Se ganhou, notifica e reinicia
    if (analise.ganhouJogo) {
        alert(jogadorDaVez == JOG_HUMANO ? 'Você venceu!' :
              'Você perdeu...');

        location.reload();
        return;
    }

    // Se não encaçapou legalmente, passa a vez
    else if (!analise.encacapou) {
        jogadorDaVez = (jogadorDaVez == JOG_HUMANO ?
                         JOG_COMPUTADOR : JOG_HUMANO );
    }

    // continua...
}
```

Também é preciso determinar a bola da vez. No caso, a de menor número ainda na mesa. Também, se for vez do computador, chamamos tacadaComputador para realizar a jogada automática:

```
function analisarJogada() {
    // ...

    // Mudar a bola de ordem
    for (var i in bolasColoridas) {
        var b = bolasColoridas[i];
```

```

    if (!b.foraDeJogo) {
        bolaOrdem = b;
        break;
    }
}

if (jogadorDaVez == JOG_COMPUTADOR)
    tacadaComputador();
}

```

Finalmente, tomemos uma providência com relação à bola 9. Se ela for encaçapada de maneira ilegítima, temos de retorná-la à mesa. Na função `encacapouColorida`, se o número da bola for 9, damos a ela o mesmo tratamento que à bola branca:

```

function encacapouColorida(bola) {
    encapapadas.push(bola);

    animacao.novoProcessamento({
        processar: function() {
            if (bola.id == 9) {
                bola.foraDeJogo = true;
                bola.corpo.SetLinearVelocity(new b2Vec2(0, 0));
                bola.corpo.SetPosition(new b2Vec2(-100/ESCALA,
                                                -100/ESCALA));
            }
            else {
                bola.tirarDoJogo();
            }
        }

        animacao.excluirProcessamento(this);
    });
}

```

Também na função `bolasPararam`, que a voltará para seu lugar inicial. Isso pode ser feito em qualquer posição da função (pode ser antes ou após a verificação da bola branca, só para agrupar lógicas semelhantes):

```
function bolasPararam() {
    var bola9 = bolasColoridas[8]; // começa em zero
    if (bola9.foraDeJogo) {
        bola9.corpo.SetPosition(
            new b2Vec2(560/ESCALA, 200/ESCALA));
        bola9.foraDeJogo = false;
    }
}

// ...
}
```

Uau! Já é possível jogar uma partida completa! E o computador é impiedoso e trapaceiro! Usa o Box2dWeb para prever o futuro!

NÍVEL DE DIFICULDADE

Você pode inserir fatores aleatórios com `Math.random` na classe `TacadaAutomatica` para fazer o computador rejeitar jogadas favoráveis de vez em quando, diminuindo assim o nível de dificuldade. Outra sugestão é fazer os testes de 10 em 10 graus (ou outro intervalo), tornando o processamento mais rápido e aumentando as chances de não encontrar uma tacada conveniente.

11.2 MOSTRADORES

Para melhorar a experiência final do jogador, vamos criar um sprite que mostra qual a bola de ordem.

Um último script para linkar:

```
<script src="js/mostrador.js"></script>
```

Em `iniciar`, instancie o sprite e insira-o na animação. Isso deve ser feito após a definição da bola de ordem:

```
function iniciar() {
    // ...

    jogadorDaVez = JOG_HUMANO;
    bolaOrdem = bolasColoridas[0];

    // Quebra aqui para criar o mostrador
    mostrador = new Mostrador(context);
    mostrador.bolaOrdem = bolaOrdem;
    animacao.novoSprite(mostrador);

    animacao.ligar();
}
```

Segue o esqueleto da classe. No construtor, passamos a constante `BOLA_SPRITE`, criada em `bola.js`, claramente criando uma dependência entre os scripts:

```
function Mostrador(context) {
    this.context = context;
    this.bolaOrdem = null;
    this.sheet = new Spritesheet(context, BOLA_SPRITE, 10, 19);
}

Mostrador.prototype = {
    atualizar: function() {
        //
    },
    desenhar: function() {
        //
    }
}
```

Imprimimos o texto "Próxima" e desenhamos a bola em tamanho reduzido:

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

```

ctx.fillStyle = 'white';
ctx.font = '10px';
ctx.fillText('Próxima:', 380, 370);

this.sheet.linha = this.bolaOrdem.id;
this.sheet.desenhar(430, 362, 10, 10);

ctx.restore();
}

```

De volta à página HTML, na função analisarJogada , atribua ao mostrador a nova bola de ordem quando houver mudança:

```

function analisarJogada() {
    // ...

    // Mudar a bola de ordem
    for (var i in bolasColoridas) {
        var b = bolasColoridas[i];

        if (!b.foraDeJogo) {
            bolaOrdem = b;
            mostrador.bolaOrdem = b;
            break;
        }
    }

    // ...
}

```

O jogo de bilhar está pronto! Obviamente, há muito espaço para melhorias e incrementos na jogabilidade, na funcionalidade, na arquitetura, que você pode implementar como forma de se exercitar e fixar seus novos conhecimentos. Não quis, nem de longe, criar o jogo perfeito, e preferi me ater ao essencial. Mas espero, com sinceridade, que tenha gostado.

Nós aprendemos a:

- capturar gestos em telas touch com o Hammer.js;
- associar corpos físicos aos sprites;
- detectar eventos de colisão e agendar a modificação ou exclusão dos corpos para depois do Step ;
- disparar callbacks para o jogo e suas classes implementar as regras de negócio nesses eventos.

SUGESTÕES PARA MELHORAR O JOGO

- Uma funcionalidade que faz sumir os controles e botões, permitindo-nos ver toda a mesa.
- Mudar de lado o controle do taco, a barra de força e o botão da tacada de acordo com o lado da mesa em que se encontra a bola branca.
- Ou, quem sabe, sumir com o controle circular do taco, permitindo-nos fazer o *rotate* em qualquer ponto da mesa. Isto certamente traria um grande ganho de jogabilidade, e só preferi fazer do modo mais restrito para demonstrar como limitar a área.
- Menus de configuração e níveis de dificuldade.
- Jogo multiplayer com Ajax e tecnologias de servidor web.
- E o que mais sua imaginação criar.

Conclusão

Chega ao fim mais uma jornada pelo fantástico mundo do Canvas do HTML5 e suas possibilidades. Nem de longe eu quis esgotar os assuntos. Acredito que a maior lição que tem de ficar de todo este esforço é que desenvolvimento de jogos, como qualquer desenvolvimento de software, requer estudo constante, uma vida dedicada a estudar e a praticar. Por isso deixei tantos links no livro, para que você possa se aprofundar da mesma forma como eu precisei me aprofundar em cada tópico. Você pode e deve saber que existe mais além do que trouxe aqui.

Estarei esperando links para os jogos feitos por meus leitores, seja online, na Play Store, na Apple Store... Divulgue suas criações para nós no fórum da Casa do Código:

<http://forum.casadocodigo.com.br/>

Bom desenvolvimento!

CAPÍTULO 12

REFERÊNCIAS BIBLIOGRÁFICAS

CÁSSIO, Éderson. Desenvolva Jogos com HTML5 Canvas e JavaScript. Casa do Código, 2014.

FERONATO, Emanuele. Box2D for Flash Games. Packt Publishing, 2012.

LOPES, Sérgio. A Web Mobile - Programe para um mundo de muitos dispositivos. Casa do Código, 2013.

ZEMEL, Tárcio Zemel. Web Design Responsivo - Páginas adaptáveis para todos os dispositivos. Casa do Código, 2013.

WARGO, John M. Apache Cordova 3 Programming. Addison-Wesley Professional, 2013.