Capítulo

4

Desenvolvimento para Dispositivos Móveis usando Tecnologias Web com Ênfase em Jogos

André Santanchè, Renoir Boulanger, Gabriela Viana, Ricardo Panaggio, Bruno Melo, Hugo Aboud

Abstract

Games are typically challenging applications to develop and they are omnipresent in the mobile platforms. Besides the hardware limitations and heterogeneity of mobiles, we are experiencing a "software platform epoch". Platforms like iOS, Android and Windows Phone offer to developers complete hermetic environments, which come with compatibility side effects, constraining applications to be reimplemented in order to address each platform. One of the most promising approaches to face this platform battle is based on the combination of Web technologies – HTML5, CSS3 and JavaScript – to produce platform-independent applications for mobiles. The question here is how to achieve the balance between independence and performance, required by games. This mini-course gives an overview of this scenario focusing in aspects addressed to game development.

Resumo

Jogos são tipicamente aplicações de desenvolvimento desafiador e são onipresentes nas plataformas de dispositivos móveis. Além das limitações e heterogeneidade do hardware em dispositivos móveis, nós estamos vivenciando uma "época das plataformas de software". Plataformas como iOS, Android e Windows Phone oferecem a seus desenvolvedores ambientes completos e herméticos, que trazem efeitos colaterais de compatibilidade, obrigando que aplicativos sejam reimplementados para atender a cada plataforma específica. Frente a este cenário, este texto apresenta uma abordagem promissora para produzir aplicativos independentes de plataforma para dispositivos móveis, a partir da combinação de tecnologias Web – HTML5, CSS3 e JavaScript. A questão aqui é como alcançar o equilíbrio entre independência e desempenho, exigido pelos jogos. Este minicurso oferece uma panorâmica deste cenário tendo como foco aspectos voltados ao desenvolvimento de jogos.

4.1. Introdução

O tema de desenvolvimento baseado em tecnologias Web sempre levanta alguma suspeita. Há dúvida sobre o desempenho dos aplicativos resultantes e da real flexibilidade na construção de interfaces exigida pelos jogos.

Porém este cenário tem mudado muito. Diversas organizações e pessoas têm se empenhado em transformar a Web em uma plataforma universal para o desenvolvimento de aplicativos, e isto inclui o universo dos dispositivos móveis. Nos últimos anos não apenas as tecnologias Web evoluíram para tornar possível o desenvolvimento de aplicativos completos, como também têm alcançado desempenho para dar suporte ao desenvolvimento de jogos populares.

Em 2013, o número de usuários acessando o Facebook por dispositivos móveis ultrapassou aqueles que acessam pela Web (Kelly, 2013). Este crescente mercado tem se segmentado em diferentes plataformas de software, definidas seja por fabricantes de hardware – e.g., iOS da Apple, BlackBerry OS – seja por companhias que desenvolvem o sistema operacional – e.g., Android da Google, Windows Phone da Microsoft. Cada plataforma define um conjunto especializado de linguagens e APIs para acesso a seus dispositivos, exigindo que desenvolvedores multipliquem seus esforços para atender a mais de uma plataforma.

O desenvolvimento de aplicativos independente de plataforma para dispositivos móveis se apresenta como uma alternativa interessante, que reduz esforços e custos. O princípio básico é desenvolver uma única vez um aplicativo capaz de se adaptar às diferentes plataformas (multiplataforma). Além da heterogeneidade destas plataformas, há desafios na crescente diversidade de dispositivos, com diferenças de resolução, capacidade e funcionalidades.

Segundo Martin Fowler (2012), quando um aplicativo multiplataforma tenta mimetizar o comportamento nativo das aplicações em cada uma das plataformas alvo, usualmente o resultado não agrada ao usuário. O problema é que uma aplicação multiplataforma nunca será capaz de reproduzir com precisão o comportamento de aplicações nativas. Deste modo, uma mimetização aproximada causa uma sensação de que algo não está funcionando tão bem. A solução apresentada por (Fowler, 2012) é portar a plataforma completa. A Web pode ser tratada como uma plataforma a ser portada, sobre a qual se desenvolvem os aplicativos. O usuário já está habituado com o modo de interagir com aplicativos Web e encontrará a mesma abordagem nos diferentes dispositivos.

A combinação entre HTML, CSS e JavaScript atingiu um grande grau de maturidade no desenvolvimento de aplicativos independentes de plataforma na Web. Enquanto a Hypertext Markup Language (HTML) é a linguagem de publicação da Web (http://www.w3.org/html/), o Cascading Style Sheets (CSS) é um mecanismo complementar para adicionar estilo a documentos Web (http://www.w3.org/Style/CSS/). Finalmente, a linguagem de programação JavaScript opera de forma integrada com o HTML e CSS.

Dentre os progressos alcançados no desenvolvimento de aplicativos, está a possibilidade de se usar estas três tecnologias para o desenvolvimento de aplicativos locais autônomos. O JavaScript originalmente desenvolvido para operar como acessório

sobre páginas passa a ser uma linguagem completa, com funcionalidades de acesso a APIs e armazenamento locais. Outro avanço, que é de especial interesse no contexto de jogos, é a possibilidade de rompimento com a interface convencional baseada em documentos, através do uso de linguagens de marcação como o SVG, que introduz um novo horizonte para a concepção de jogos Web.

Esta Web como plataforma de desenvolvimento local é o objeto central de estudo deste texto, que tem enfoque em sua aplicação no desenvolvimento para dispositivos móveis.

A estrutura do texto está organizada da seguinte maneira: na Seção 4.2 são revisados conceitos chave das tecnologias Web em questão, com enfoque no desenvolvimento de aplicativos para dispositivos móveis, bem como são apresentados avanços importantes destas tecnologias que contribuem para este desenvolvimento; na Seção 4.3 são descritas tecnologias para a construção de gráficos e animação que são chave no desenvolvimento de jogos; na Seção 4.4 são tratados alguns fundamentos importantes de JavaScript para o desenvolvimento de aplicativos; na Seção 4.5 são apresentadas estratégias e boas práticas para o uso de marcadores de forma eficiente e reusável; na Seção 4.6 são sistematizados alguns conceitos sobre os Web Components; na Seção 4.7 são descritas as diferentes arquiteturas que dão suporte ao desenvolvimento com tecnologias Web em dispositivos móveis; na Seção 4.8 é dada uma visão geral de alguns fundamentos para o projeto de aplicativos com foco no usuário; na Seção 4.9 serão apresentadas as considerações finais.

4.2. Visão geral de HTML5 e CSS3 para Aplicativos

Esta seção dá uma visão geral de alguns conceitos das tecnologias HTML5 e CSS3 que são importantes para este texto. Por serem temas amplamente conhecidos, com uma vasta documentação disponível, esta seção parte do pressuposto de que o leitor já conhece fundamentos de HTML e CSS. Uma versão mais aprofundada desta seção está disponível em nosso texto anterior (Santanchè, 2013).

4.2.1. HTML5 e CSS3

A especificação da linguagem HTML é mantida pelo World Wide Web Consortium (W3C). Desde a sua última versão 4, o W3C vem se empenhando na especificação da sua versão sucessora, conhecida como HTML5. Entretanto, tal especificação assumiu um novo caráter de "*living standard*" (Berjon et al., 2012) pela participação da organização WHATWG (http://www.whatwg.org). Living se refere ao fato de que uma especificação HTML constantemente atualizada é mantida pelo WHATWG, de modo que inovações possam ser adotadas e experimentadas dinamicamente, na medida em que o padrão evolui, sem compartimentar saltos de evolução em versões. O W3C continua responsável pelo padrão, lançando especificações versionadas.

HTML5, portanto, tornou-se uma "*buzzword*" que incorpora os mais recentes avanços do HTML após a versão 4, com funcionalidades de semântica, acessibilidade, recursos multimídia e de interação, que só eram possíveis com o uso de tecnologias de terceiros (*Flash like*). O HTML5 inaugurou uma nova relação com aplicativos, oferecendo-lhes acesso a funcionalidades que antes exigiam estratégias indiretas, complexas e eventualmente extensões não padronizadas à linguagem. Para dar suporte

as novas funcionalidades como persistência, desenhos 2D, áudio e vídeo, estão sendo desenvolvidas novas APIs.

O CSS permite separar a forma com que os elementos HTML deveriam ser exibidos do seu conteúdo. Esta separação conteúdo-estilo é fundamental no desenvolvimento de aplicações que se adaptam a vários dispositivos. As especificações do CSS são definidas em níveis ao invés de versões, em que cada nível acrescenta funcionalidades ao anterior. O CSS nível 3, CSS3, dividiu o processo de especificação em pequenos módulos que evoluem de forma independente, com base a especificação CSS 2.1. Além disto, adicionou funcionalidades para efeitos de texto, transformações 2D/3D e animações.

4.2.2. Separação Conteúdo – Estilo

Ao se tratar de layout no contexto de dispositivos móveis, uma questão chave é adaptabilidade, dada a diversidade de formatos e resoluções. A Figura 4.1 ilustra o aspecto chave para tratar a questão: a separação de conteúdo e estilo, usando a semântica como ponte.

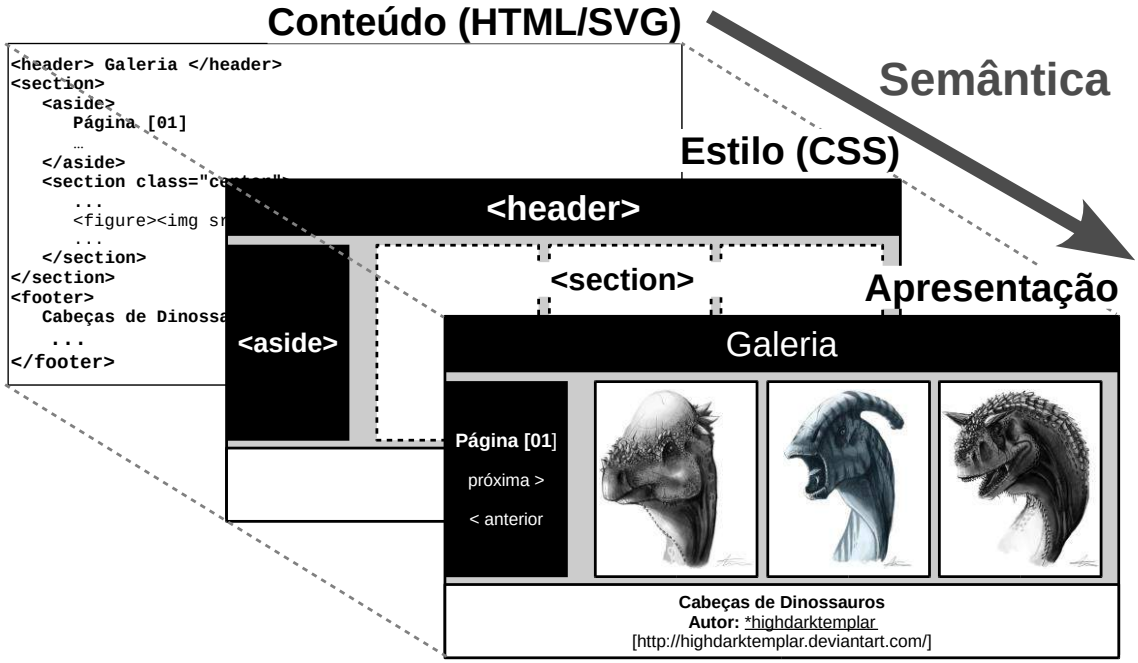


Figura 4.1. Separação Conteúdo-Estilo em uma tela de aplicativo.

Na figura, o Conteúdo representa os elementos da tela inicial de um aplicativo. Eles estão dispostos em categorias, conforme seu papel, que são anotações semânticas sobre os dados. Uma segunda camada de Estilo associa elementos anotados semanticamente a seu layout visual. O Conteúdo associado às diretrizes de apresentação do Estilo produz a Apresentação, ilustrada na terceira camada. Deste modo, o mesmo conteúdo fonte pode ser associado a diferentes diretrizes de apresentação para se adaptar a diferentes contextos.

Em nosso texto anterior (Santanchè, 2013) tratamos com detalhes esta questão, apresentando técnicas e melhores práticas de realizar tal separação e como o HTML5 tem progredido na direção de uma marcação mais semântica.

4.2.3. Responsividade e Adaptabilidade

Em 2010, Ethan Marcotte (2010) chama atenção para uma disciplina emergente na arquitetura chamada *Responsive Architecture*. A questão central é projetar ambientes que respondem às mudanças e à presença ou ação de pessoas, explorando sensores, robótica e novos materiais para criar paredes flexíveis que se ajustam; divisórias de smart glass que se tornam opacas quando pessoas iniciam uma reunião; ajustes de luz e temperatura que se adequam à presença e volume de pessoas (Marcotte, 2010). O termo Responsive Design deriva desta inspiração, em que Marcotte propõe que "Ao invés de customizar projetos desconexos para cada e sempre crescente número de dispositivos web, nós podemos tratá-los como facetas da mesma experiência"[1] (Marcotte, 2010). Algumas novidades introduzidas pelo HTML5 e CSS3 – a exemplo da media query e dos layouts flexíveis – contribuem na construção de interfaces Web "responsivas". Mas antes disto, deve-se considerar quais as características que se esperam neste tipo de projeto adaptável/responsivo.

Um exemplo bastante característico desta estratégia está ilustrado na Figura 4.2. Trata-se do Table:Reflow do framework jQuery (http://view.jquerymobile.com/1.3.2/dist/demos/widgets/table-reflow/). Este tipo de componente é capaz de ir adaptando o formato da tabela de acordo com o layout visual e proporções do dispositivo em que ela é apresentada. No limite, em telas muito estreitas (Figura 4.2 direita), o table reflow transforma a tabela em uma lista de "cartões". Com isto o usuário restringe o seu "scroll" a uma única dimensão, facilitando a acesso a dados em dispositivos com capacidade de apresentação limitada.

Rank	Movie Title	Year	Rating	Reviews
1	Citizen Kane	1941	100%	74
2	Casablanca	1942	97%	64
3	The Godfather	1972	97%	87
4	Gone with the Wind	1939	96%	87
5	Lawrence of Arabia	1962	94%	87
6	Dr. Strangelove Or How I Learned to Stop Worrying and Love the Bomb	1964	92%	74
7	The Graduate	1967	91%	122
8	The Wizard of Oz	1939	90%	72
9	Singin' in the Rain	1952	89%	85
10	Inception	2010	84%	78

Rank	1
Movie Title	Citizen Kane
Year	1941
Rating	100%
Reviews	74
Rank	2
Movie Title	Casablanca
Year	1942
Rating	97%
Reviews	64

Figura 4.2. Exemplo de Responsive Design no Table Reflow.

Em nosso texto anterior (Santanchè, 2013) tratamos em detalhes algumas técnicas que permitem a criação de design adaptável/responsivo, inclusive revisando

[1] Tradução livre (Marcotte, 2010)

alguns conceitos básicos de CSS no projeto de layout. Dentro as inovações do CSS3, são apresentados os conceitos de *flexible box* ou flexbox (Atkins Jr. et al., 2012) – que consiste em uma técnica baseada em um gerenciamento automatizado de layout – e *media query* (Lie et al., 2012) que permite inspecionar o contexto e estabelecer escolhas condicionais de estilo.

4.2.4. Semântica nos documentos Web

Desde o começo da Web, páginas apresentadas ao usuário mantiveram suas raízes no HTML, muito embora a Web tenha evoluído para padrões mais genéricos, como a linguagem XML. A Extensible Markup Language – XML (Bray et al., 2008) foi criada com o intuito de permitir a uma comunidade definir seus próprios marcadores, as regras de composição de documentos e, consequentemente, o modo como os segmentos marcados serão interpretados. Isto a classifica como uma metalinguagem (linguagem para a definição de linguagens), ao contrário do HTML que possui um conjunto predefinido de marcadores.

Iniciativas como Microformats (Khare, 2006), RDFa (Adida & Birbeck, 2008) (Adida et al., 2011) e Microdata (Hickson, 2011) perceberam a importância de permitir que usuários definam seus próprios vocabulários (como acontece no XML), de modo que possam criar anotações em páginas HTML, sem romper com a abordagem nativa de representação do conteúdo. As anotações cumprem o papel de metadados e permitem uma interpretação mais rica do conteúdo por máquinas e humanos. Neste sentido, elas enriquecem semanticamente este conteúdo.

Em nosso texto anterior (Santanchè, 2013) apresentamos a iniciativa dos Microformats que foi pioneira neste contexto. Neste texto, apresentaremos o RDFa. Enquanto o Microformats define uma estratégia para incorporar vocabulários em documentos HTML, a interpretação da semântica dos mesmos é delegada para os programas, já que ela não pode ser descrita de forma explícita usando o próprio Microformats. Neste sentido, ele está bastante alinhado com a possibilidade de construção de vocabulários XML.

O RDFa, por outro lado, está alinhado com a camada de semântica explícita da Web Semântica, o *Resource Description Framework* – RDF. O RDF define um modelo e uma linguagem para a representação homogênea de descrições associadas a recursos, que podem ser identificados por meio da Web (Manola, 2004). Ele á capaz de tornar a semântica das descrições explícita através de uma rede de relações semânticas entre conceitos, identificados através das URIs. Por esta razão, o RDF é a base, por exemplo, para a construção de ontologias.

O RDFa torna possível mesclar descrições RDF em documentos HTML. A descrição detalhada da Web Semântica e do RDF vão além do escopo deste material. Sumarizaremos aqui alguns conceitos importantes para a autoria na Web, dando uma visão geral de como implementá-los.

Considere que queremos representar em RDF a seguinte sentença: “o céu tem cor azul”. O RDF representa tais sentenças em um modelo baseado em triplas, que explicita a semântica da descrição. Cada tripla define o recurso que está sendo descrito que está sendo descrito – o “céu” neste exemplo – uma propriedade – a “cor” neste exemplo – e o valor para a propriedade – “azul” neste exemplo. Isto forma a tripla

(recurso, propriedade, valor) com os valores (“céu”, “cor”, “azul”). Os elementos da tripla, aqui apresentados de forma abstrata entre aspas, serão na maioria dos casos entidades representadas de forma única por URIs e dentre elas conceitos em ontologias.

O exemplo a seguir mostra uma reinterpretação das receitas sob a ótica do RDFa. O atributo especial `property` serve para mediar a construção de triplas. Ele associa um recurso, definido pelo bloco em que está inserido, com uma propriedade declarada em `property` e um valor, que é o valor do elemento que tem o atributo `property`. Também é usado um atributo do tipo `typeof`, que define recursos que são instâncias de classes.

```
<div xmlns:v="http://rdf.data-vocabulary.org/#"
     typeof="v:Recipe">
...
<p property="v:yield">6 to 8 servings</p>

...
<ul rel="v:ingredient">
  <li typeof="v:Ingredient">
    <span property="v:amount">1/2 pound</span> of
    <span property="v:name">elbow macaroni</span></li>
...
</ul>
...
<p property="v:instructions">
<p>Preheat oven to 350 degrees F.</p>
...
</div>
...
</div>
```

Se por um lado um documento HTML pode ser visto como a base para construção de aplicativos, tal como estamos tratando neste documento, através do RDFa ele se torna base para a construção de descrições semânticas aptas a serem extraídas e recuperadas na forma de query.

O query sobre RDF pode ser feito através do SPARQL (W3C SPARQL Working Group, 2013), que pode ser definido de forma simplificada como uma linguagem de *query* no estilo SQL, mas voltado para o modelo RDF de triplas e rede de associações.

4.3. Gráficos vetoriais, SVG, Canvas e WebGL

Tecnologias Web têm permitido ir além do clássico modelo de construção visual baseada em documentos, para a exploração de um espaço vetorial, usado não apenas para a inserção de imagens dentro de documentos, mas para o projeto da interface completa. Isso é especialmente importante no desenvolvimento de jogos e, por esta razão, damos especial atenção ao tema nesta seção.

Nos primórdios do desenvolvimento Web, o layout das páginas era composto basicamente por textos e imagens dispostos linearmente. Eventualmente eram usados frames ou tabelas delimitando e organizando seções dentro da página. Tal paradigma foi se tornando obsoleto com o surgimento do CSS e a consequente possibilidade de se posicionar blocos de conteúdo associado a estilo, definidos pelas tags `<DIV>`.

Porém, o grande avanço no sentido de dinamização de conteúdo veio com a possibilidade de alterar propriedades de estilo (CSS) dos elementos após o carregamento da página, através de Javascript. Frameworks como o jQuery (http://jquery.com) tornaram a Web uma experiência mais interativa, apesar de ainda limitada pelo conceito de blocos de conteúdo associados aos DIVs, que exigiam estratégias indiretas para a construção de primitivas gráficas, não previstas originalmente no HTML e que afetavam o bom desempenho em navegadores.

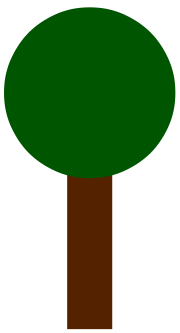
Para projetos de interfaces mais arrojadas, que façam uso de formas mais complexas, as DIVs podem dificultar o desenvolvimento, dada a sua natureza quadrada. Pode-se utilizar estilos em CSS3 e imagens transparentes para talhar forma e efeitos desejados, porém, muita memória e processamento são utilizados para a apresentação.

Além disso, se o projeto exigir muitos elementos em uma mesma cena, a utilização de DIVs poderá apresentar desempenho insatisfatório na experiência do usuário. Por este motivo, existe uma tendência de não se utilizar as DIVs como solução, em projetos preocupados com questões de desempenho e fluidez da experiência de usuário.

4.3.1. SVG

Um avanço neste sentido foi o suporte nativo dos navegadores ao formato SVG – Scalable Vector Graphics (Dahlström et al., 2011). O SVG nasceu como um formato XML para a representação de imagens vetoriais. Enquanto formatos raster – tais como GIF, JPG e PNG – representam a imagem como uma matriz de pixels, formatos vetoriais representam a imagem como uma coleção de objetos geométricos, em que são registradas propriedades de posicionamento, dimensões e características de apresentação, como cor e transparência. A imagem é reconstruída em cada apresentação. Imagens vetoriais se adaptam a diferentes configurações de apresentação, sendo capazes de explorar os limites de resolução da área de apresentação.

Ao contrário de outros padrões vetoriais, como o WMF e EPS, o SVG representa a imagem em formato texto baseado em XML. Por esta razão ele tem se tornado um padrão para apresentar conteúdo vetorial na Web. É possível inserir diretamente uma imagem SVG dentro de um documento HTML. A Figura 4.3 ilustra um segmento SVG tal como inserido no corpo (`<body>`) de uma documento HTML.



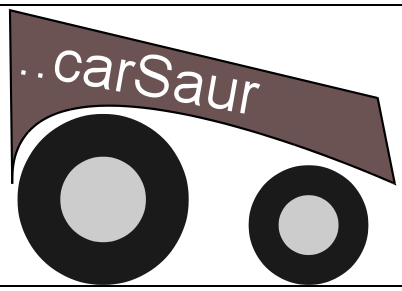
```
<svg id="desenho" version="1.1"
     xmlns="http://www.w3.org/2000/svg"
     width="205" height="370">

    <rect style="fill:#552200"
          x="77" y="179" width="50" height="190" />

    <circle style="fill:#005500"
            cx="102" cy="106" r="95" />

</svg>
```

Figura 4.3. Imagem de uma árvore descrita em SVG.

O tag `<svg>` delimita e dimensiona (através dos atributos `width` e `height`) a área de desenho. Dentro desta área foram usadas as seguintes primitivas de desenho:

Primitiva	Descrição	Atributos	
`<rect>`	Desenha um retângulo.	`style`	Estilo de apresentação. Neste caso define a cor de preenchimento.
		`x, y`	Coordenadas do canto esquerdo superior.
		`width, height`	Altura e largura do retângulo.
`<circle>`	Desenha um círculo.	`style`	Estilo de apresentação. Neste caso define a cor de preenchimento.
		`cx, cy`	Coordenadas do centro do círculo.
		`r`	Raio do círculo.

O SVG também pode ser usado para a criação de curvas mais complexas, como o carro ilustrado na Figura 4.4. Neste caso, foi usado um recurso de construção de polígonos baseado na descrição do contorno (`<path>`). Além disto, há um texto rotacionado.




```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg"
     width="181" height="130">

  <circle style="fill:#1a1a1a" cx="44" cy="90" r="40" />
  <circle style="fill:#cccccc" cx="44" cy="90" r="20" />

  <circle style="fill:#1a1a1a" cx="140" cy="102" r="28" />
  <circle style="fill:#cccccc" cx="140" cy="102" r="14" />

  <path style="fill:#6c5353; stroke:#000000; stroke-width:1px"
        d="M 180.322,82.637687 172.30769,42.566127 0.50088787,1.4927774
           1.5026779,82.637677 c -2.50447,-82.14667965
           178.8193221,1e-5 178.8193221,1e-5 z" />

  <text style="fill:white; font-size:28px; font-family:Arial"
        x="9" y="30"
        transform="rotate(10)">
    ..carSaur
  </text>


</svg>
```

Figura 4.4. Imagem de um carro descrito em SVG.

Para a construção deste carro foram usadas as seguintes primitivas adicionais:

Primitiva	Descrição	Atributos	
`<path>`	Descreve um trajeto que usualmente será usado para a definição de contornos de polígonos.	`style`	Estilo de apresentação. Neste caso define a cor de preenchimento e do contorno e a espessura do contorno.
		`d`	Sequência de contorno formada por letras que representam primitivas de descrição do contorno e coordenadas.
`<text>`	Insere um texto.	`style`	Estilo de apresentação. Neste caso define a cor e fonte da letra.
		`x, y`	Coordenadas do esquerdo inferior do texto.

O atributo `transform`, associado neste exemplo à primitiva `<text>` permite a aplicação de transformações geométricas sobre as primitivas. Tal atributo pode ser aplicado na maioria das primitivas e permite a realização de transformações de: rotação (`rotate`), translação (`translate`), mudança de escala (`scale`) etc.

O SVG não apenas pode ser inserido em páginas HTML, mas pode assumir o lugar do HTML na definição da estrutura do documento. Ou seja, é possível construir um aplicativo Web completamente a partir de uma descrição SVG. Isto permite romper o layout tipicamente horizontal e orientado a documentos do HTML, o que é essencial no desenvolvimento de jogos. O SVG interage com os demais atores de um documento Web do mesmo modo que o HTML. É possível se associar folhas de estilo ao SVG, bem como scripts em JavaScript.

O exemplo a seguir mostra uma redefinição do exemplo anterior, com o uso de uma folha de estilos associada ao SVG. Do lado esquerdo, está a descrição SVG e do lado direito a folha de estilo usada pela descrição.

```
<svg version="1.1"
xmlns="http://www.w3.org/2000/svg"
     width="181" height="130">
  <circle class="tyre" cx="44" cy="90" r="40" />
  <circle class="rim" cx="44" cy="90" r="20" />

  <circle class="tyre" cx="140" cy="102" r="28"/>
  <circle class="rim" cx="140" cy="102" r="14" />

  <path class="frame"
        d="M 180.322,82.637687 172.30769,42.566127
          0.50088787,1.4927774 1.5026779,82.637677
          c -2.50447,-82.14667965 178.8193221,1e-5
          178.8193221,1e-5 z" />

  <text class="nameStyle" x="9" y="30"
        transform="rotate(10)">
    ..carSaur
  </text>

</svg>
```

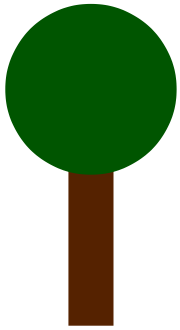
```
<style type="text/css">
  .tyre {
    fill: #1a1a1a;
  }
  .rim {
    fill: #cccccc;
  }
  .frame {
    fill: #6c5353;
    stroke: #000000;
    stroke-width: 1px;
  }
  .nameStyle {
    fill: white;
    font-size: 28px;
    font-family: Arial;
  }

</style>
```

4.3.2. Canvas

A presença de um renderizador vetorial nativo nos navegadores – trazido pelo SVG – permitiu a concepção de um mecanismo no HTML5 que o utilizasse de forma dinâmica para gerar conteúdo: o canvas. Esta nova abordagem substitui o mecanismo de blocos de conteúdo, introduzindo a possibilidade de construções mais complexas e dinâmicas.

O canvas surgiu como uma ferramenta da Apple em seu WebKit e passou a ser adotado pela W3C como uma forma padrão de criação de imagens diretamente em uma área do navegador, pelo uso de primitivas de desenho. O canvas se comporta como uma tela de pintura, em que programas em JavaScript podem plotar coisas. Ele é capaz de traçar desde simples quadrados a cenários tridimensionais complexos.

A Figura 4.5 apresenta a mesma árvore da figura anterior descrita através do uso do canvas e comandos JavaScript. No corpo do documento HTML a área do canvas é delimitada, dimensionada e nomeada pelo tag <canvas>.



```
function desenhar() {
   var desenho = document.querySelector("#desenho");

   if (desenho.getContext) {
      var contexto = desenho.getContext("2d");

      contexto.fillStyle = "#552200";
      contexto.fillRect(77, 179, 50, 190);

      contexto.fillStyle = "#005500";
      contexto.arc(102, 106, 95, 0, 2*Math.PI);
      contexto.fill();
   }
}
```

```
<canvas id="desenho" width="205" height="370">
  Área de desenho
</canvas>
```

Figura 4.5. Imagem de uma árvore construída em um canvas.

No módulo JavaScript que desenha a árvore, o objeto canvas é recuperado através de um querySelector e é recuperado um objeto de contexto bidimensional (“2d”) deste canvas. Tal objeto define um conjunto de métodos que são primitivas para desenho no canvas. Foram usados os seguintes métodos:

Atributo	Descrição		
`fillStyle`	Define o estilo de preenchimento das figuras que serão traçadas subsequentemente. No exemplo, a cor de preenchimento.		
Método	**Descrição**	**Parâmetros**	
`fillRect`	Desenha um retângulo preenchido.	`x, y`	Coordenadas do canto esquerdo superior.
		`width, height`	Altura e largura do retângulo.
`arc`	Define uma forma geométrica a ser usada posteriormente para desenho.	`cx, cy`	Coordenadas do centro do arco.
		`r`	Raio do arco.
		`começa, termina`	Indica o ângulo em que o desenho do arco começa e termina respectivamente.
`fill`	Preenche uma forma geométrica. No exemplo, o arco.		

Há diferenças fundamentais em traçar imagens em SVG ou desenhando no canvas. Uma imagem SVG é preservada vetorialmente na árvore DOM de objetos que compõem o documento. Ele será adaptado e redesenhado cada vez que a área de apresentação sofrer mudanças. O desenho no canvas funciona como riscar um quadro. Apesar de terem sido usados métodos que se comportam como as primitivas de desenho do SVG – ou seja, `fillRect()` corresponde ao `<rect>`; `arc()` mais `fill()` correspondem ao `<circle>` – uma vez que a forma é "riscada" no canvas, ela se torna uma conjunto de pixels e perde-se a sua origem de objeto vetorial.

Mesmo com a introdução do canvas, há aspectos práticos que ainda herdam características clássicas de apresentação Web, que em sua maioria não se modificam com grande dinâmica – e.g., animações. Muito se discute, no entanto, acerca do seu desempenho: trata-se de uma biblioteca gráfica "*immediate mode*", ou seja, o canvas é atualizado conforme as chamadas às funções de desenho ocorrem, não havendo um buffer que esconde do usuário as etapas do render (*double-buffering*). Devido a este problema, em imagens de alta complexidade o usuário pode ver o '*flick'* da tela sendo atualizada. Há várias formas de contornar este problema – como, por exemplo, o uso de display lists (canvas pré-renderizados que são inseridos na cena em runtime) e buffers de hardware – mas, em termos práticos, o método ainda se mostra ineficaz para aplicações mais complexas.

Visando melhorar desempenho de aplicações 3D na Web, o WebGL (http://webgl.org) surgiu como um aprimoramento do canvas. Para isto utilizou-se da recente comunicação com hardware que os navegadores passaram a oferecer. A comunicação mais direta com a GPU permitiu a renderização de buffers 3D em tempo satisfatório, em comparação com o canvas, utilizando-se do conceito de matrizes de

projeção e model-view, presentes no OpenGL. A Graphics Processing Unit (GPU) é uma unidade de processamento especializada no tratamento dos gráficos que são apresentados no display. Usualmente, ela fica situada na placa gráfica do computador.

A comunicação com a GPU permitiu, além do ganho de desempenho, a implementação de shaders (scripts que rodam na GPU) específicos para o aplicativo que está em desenvolvimento e a possibilidade de trabalhar com muito mais flexibilidade em cores e texturas, bem como na implementação mais eficaz de interfaces 3D. Isto representa um grande avanço no desenvolvimento de jogos Web/mobile, uma vez que apresenta uma forma robusta e flexível de renderização.

Já existem centenas de exemplos didáticos da capacidade das novas formas de renderização do HTML5 espalhados pela Web, utilizando-se de técnicas de canvas e WebGL, com e sem o uso de frameworks. Muitas vezes misturam-se técnicas no desenvolvimento, como por exemplo jogos cuja interface 3D é desenvolvida em WebGL e a HUD (Heads-up Display, informações que aparecem em posição estática sobre o jogo) através de DIVs. No entanto, por ser uma tecnologia nova, pouco se explorou comercialmente de suas possibilidades.

4.3.3. Animação em SVG

Há duas estratégias fundamentais para se produzir animações em SVG. A primeira envolve o uso de primitivas do próprio SVG e a segunda é feita através de programação e requer interação com o JavaScript.

Para a primeira estratégia o SVG define primitivas de animação (Dahlström, 2011), dentre as quais destacamos:

Primitiva	Descrição	Atributos	
`<animate>`	Varia um valor de atributo dentro de um período de tempo.	`attributeName`	Nome do atributo afetado.
		`from`	Valor de origem da variação.
		`to`	Valor de destino da variação.
`<animateTransform>`	Realiza uma transformação geométrica dentro de um período de tempo.	`type`	Tipo de transformação: e.g., `rotate`, `translate`.
		`from`	Valor inicial da transformação.
		`to`	Valor final da transformação.
`<animateMotion>`	Move-se por um trajeto em um período de tempo.	`path`	Trajeto a ser seguido. É descrito da mesma maneira que uma linha poligonal.

Para as três primitivas de animação valem os atributos:

Atributo	Descrição
`begin`	Instante no tempo em que a ação inicia.
`dur`	Duração da ação.
`fill`	Recebe o valor `freeze` para indicar que o objeto se manterá na posição final após a animação e recebe `remove` caso o objeto deva retornar à posição original depois da animação.

O seguinte trecho realiza uma animação aumentando o raio (atributo `r`) da jante do carro de `0` para `20` em seis segundos. Note que a primitiva de animação é colocada dentro do elemento animado:

```
<circle style="fill:#cccccc" cx="44" cy="90" r="0">
  <animate attributeName="r" attributeType="XML"
           begin="0s" dur="6s" fill="freeze" from="0" to="20" />

</circle>
```

O seguinte trecho realiza uma animação rotacionando o nome do carro de `-90` até `10` graus em seis segundos:

```
<text style="fill:white; font-size:28px; font-family:Arial"
      x="9" y="30">
  ..carSaur
  <animateTransform attributeName="transform" attributeType="XML"
       type="rotate" from="-90" to="10"
       begin="0s" dur="6s" fill="freeze" />

</text>
```

A animação por trajeto exige a definição de um caminho (`path`) na forma de linha poligonal. No exemplo a seguir a descrição geométrica do carro foi agregada dentro de um elemento de agrupamento `<g>`. O `<animatedMotion>` foi aplicado ao `<g>` de modo que todo o carro participe da animação.

```
<g>
  <circle … />
  <circle … />
  <circle … />
  <circle … />
  <path … />
  <text … />

  <animateMotion path="m 2.8571401,475.21933 c 162.8571499,-105.71428
282.8043099,34.17323 388.5714299,0 105.76713,-34.17323 127.12446,-162.77867
225.71429,-162.85714 98.58983,-0.0785 254.28571,182.85714
254.28571,182.85714"
                 begin="0s" dur="10s" fill="freeze" />
</g>
```

O resultado das animações acima está ilustrado na Figura 4.6, que além disto inclui a plotagem da linha poligonal equivalente ao trajeto a ser seguido pelo carro. É importante ressaltar que o navegador aqui carregou diretamente o arquivo SVG.

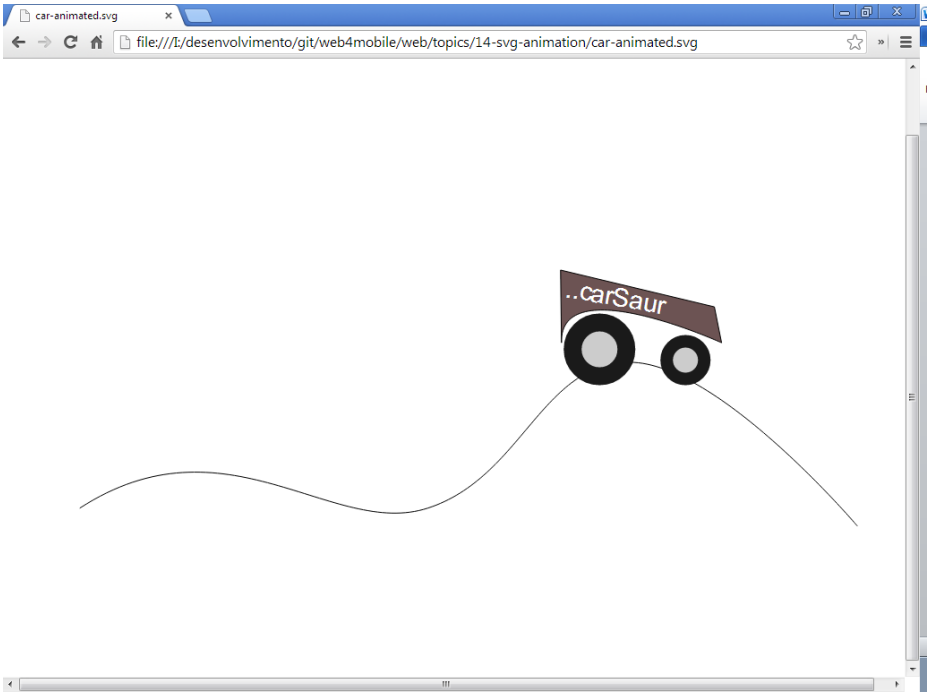


Figura 4.6. Carro executando uma animação SVG.

4.3.4. Ferramentas para SVG

Por ser o SVG um formato de gráficos vetoriais há ferramentas para a construção de imagens e paths, que podem ser exploradas no desenvolvimento de aplicativos. Dentre elas, o Inkscape (http://inkscape.org/) se destaca por editar nativamente SVG. Tais ferramentas se tornam essenciais, principalmente para a construção de curvas complexas usadas tanto nas ilustrações quanto nas animações. O ponto negativo é que nem sempre o Inkscape produzirá uma representação simples e limpa da imagem.

A Figura 4.7 ilustra um trajeto o trajeto aplicado na animação do carro sendo modelado no Inkscape pelo uso de curvas de Bezier. Após a gravação do arquivo SVG, o trajeto, representado no atributo path, foi copiado para o elemento de animação `<animateMotion>`, de modo que o carro se deslocasse por sobre o trajeto.

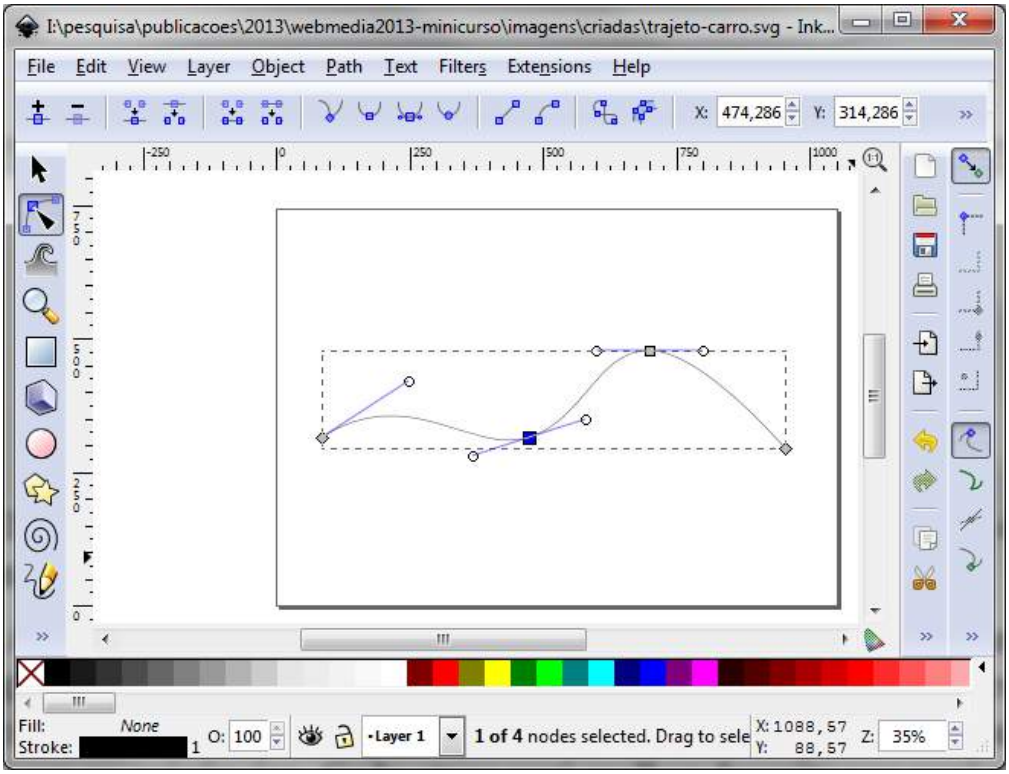


Figura 4.7. Trajeto (path) sendo modelado no Inkscape.

4.4. JavaScript para Aplicativos e DOM

JavaScript é a principal linguagem para programação em navegadores Web. Apesar de ter sido originalmente projetada para o desenvolvimento de pequenos scripts – que permitem definir o comportamento dos elementos do HTML, manipulando suas características, alterando as propriedades e valores definidos no CSS – a linguagem se tornou a melhor opção para o desenvolvimento de aplicações independentes de plataforma sobre navegadores Web. Com o HTML5 e as novas APIs, surgem mais possibilidades de controle dos elementos criados no código.

A linguagem foi inventada por Brendan Eich na Netscape e implementada no navegador Netscape 2.0. Em seguida, foi incorporada no navegador Internet Explorer 3.0 da Microsoft com o nome de JScript (Ecma, 2011). Com o intuito de padronizar a linguagem JavaScript, que passou a ser adotada e modificada nos mais diversos navegadores, a organização Ecma International lançou o padrão ECMAScript, baseado no JavaScript e JScript (Ecma, 2011). Os diversos navegadores têm progressivamente suportado o padrão ECMAScript.

O W3C constituiu o Web Applications Working Group – WebApps WG (http://www.w3.org/2008/webapps/) com o intuito de desenvolver especificações para dar suporte ao desenvolvimento de Aplicativos Web (WebApps). As especificações deste grupo serão de especial interesse nesta seção. Dentre elas, trataremos do DOM, Web Components, Web Storage e padronização das Widgets.

4.4.1. DOM - Document Object Model

O paradigma orientado a objetos tem se mostrado uma excelente ponte entre os universos de documentos (HTML/CSS/SVG) e programas de computador (JavaScript). Árvores de objetos usadas para modelar estruturas de dados neste paradigma se casam de forma elegante com árvores de conteúdo, que são representações típicas de documentos. O padrão de interfaces entre aplicativos e documentos chamado DOM – *Document Object Model* – proposto pelo W3C (Kesteren et al., 2012) tira vantagem desta similaridade.

A interface DOM é certamente um dos padrões mais importantes para se estabelecer a ponte entre o HTML/CSS/SVG e o JavaScript. No texto anterior (Santanchè, 2013) mostramos como o DOM interage com um documento HTML. O DOM não está restrito a documentos HTML/CSS, ele pode ser igualmente usado para documentos XML. Ele é fundamental na manipulação de documentos XML que o cliente intercambia com o servidor.

Mostraremos aqui como esta interação ocorre com o SVG do carro apresentado na Seção 4.1, cuja árvore DOM é ilustrada na Figura 4.8. A representação SVG fica completamente subordinada ao nó `<svg>`, situado na raiz da árvore. Como mostra a figura, os atributos `width` e `height` determinam a extensão horizontal e vertical da ilustração. Abaixo do nó raiz, aparecem: quatro nós `<circle>` que representam as duas rodas; um nó `<path>` traçando o polígono que constitui o chassi do carro; um no `<text>` contendo o nome do veículo.

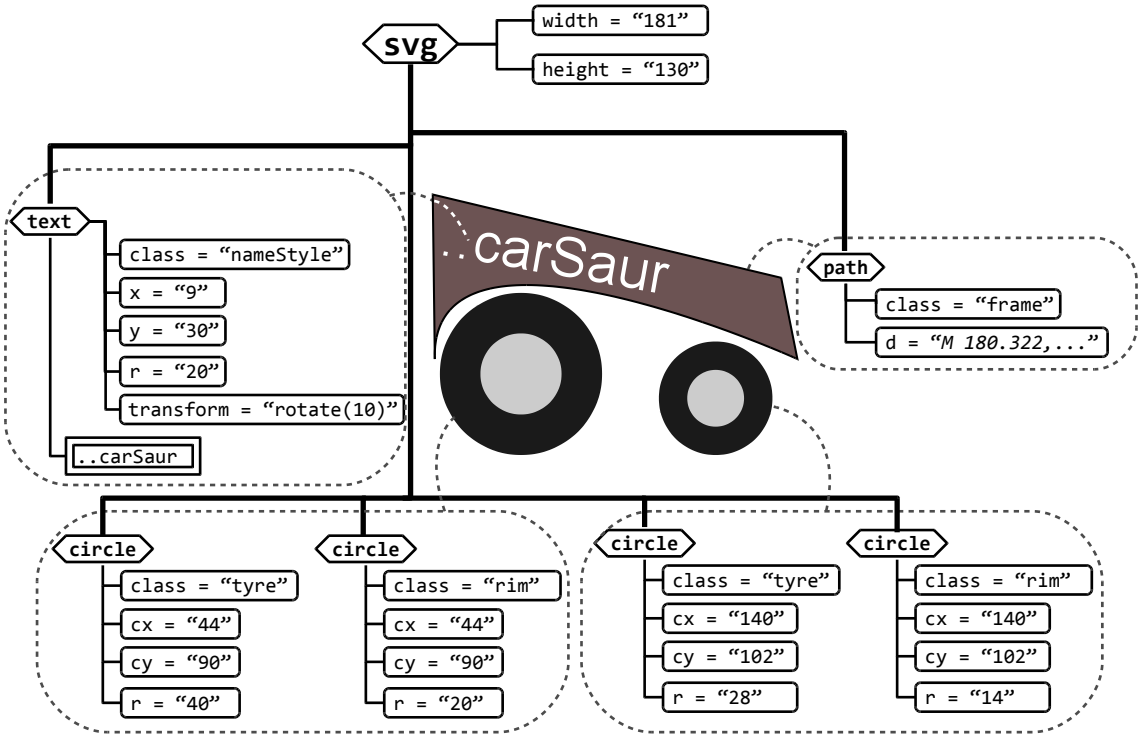


Figura 4.8. Árvore DOM do carro em SVG.

Como está ilustrado na Figura 4.9, o DOM pode ser entendido como uma lente entre o documento e o aplicativo JavaScript. Ele permite que este aplicativo visualize e

manipule tanto o conteúdo SVG quanto o estilo CSS como árvores de objetos em JavaScript.

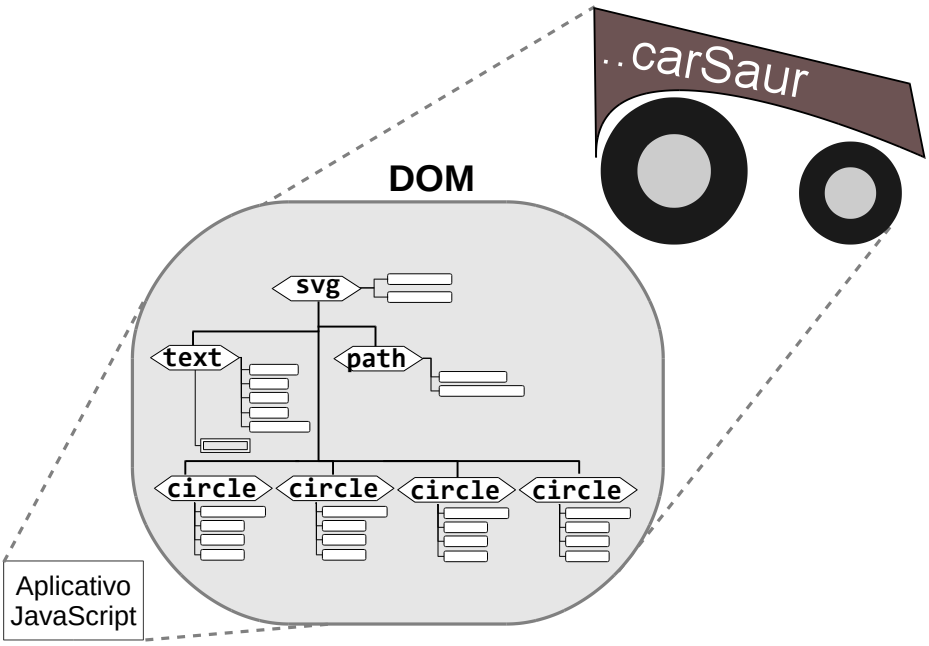


Figura 4.9. Perspectiva DOM sobre um desenho em SVG.

A especificação DOM é independente de plataforma. Por este motivo, ela não possui qualquer descrição de como seus métodos serão implementados, se limitando à interface. A implementação é delegada para soluções sobre plataformas e linguagens específicas.

Cada tipo de nó de um documento HTML ou XML se torna em uma instância de classe DOM em JavaScript. Alguns destaques:

- **Node** – Representa genericamente qualquer nó da árvore e encapsula o acesso a informações comuns a todos os nós: tipo, nome, valor, atributos etc. Também concentra os métodos fundamentais de acesso a dados, navegação e modificação da árvore. As classes a seguir são especializações de Node.
- **Element** – Elementos HTML/XML, que são representados por tags, e.g., `<text>`, `<path>`, `<circle>` etc.
- **Attr** – Atributos associados a elementos, e.g., `cx`, `cy` e `r` associados a `<circle>`.
- **Text** – Conteúdo texto livre dentro do documento.
- **Document** – Nó raiz da árvore que representa o documento completo.

Atributos disponíveis em cada nó (instâncias de Node) são usados para a navegação pelo documento. A Figura 4.10 apresenta a árvore DOM da Figura 4.8 e algumas operações de navegação. Através do método getElementById() do objeto document, é possível se alcançar diretamente um elemento a partir do valor de seu identificador. O atributo firstChild mantém um ponteiro para o seu primeiro nó filho, se houver. O atributo nextSibling permite alcançar seu nó irmão – e.g., navegar de um nó

`<circle>` para o seu vizinho. Através destes dois atributos, é possível se percorrer uma árvore inteira.

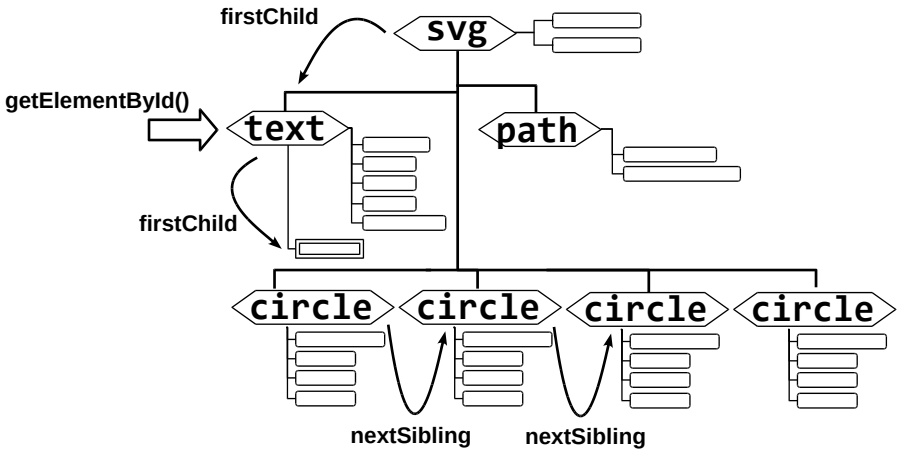


Figura 4.10. Navegação pela árvore DOM do SVG.

4.4.2. Selector

Embora o método `getElementById()` seja amplamente usado para o acesso de um elemento a partir do seu identificador, há uma nova maneira de se alcançar um elemento específico em uma árvore DOM. Trata-se da API Selector (Kesteren, 2013) que dispõe de uma sintaxe compacta e poderosa para a seleção de nós, alinhada com a sintaxe do selector do CSS e similar a usada pelo jQuery em sua biblioteca selector (Sizzle).

Considere o trecho HTML a seguir que será usado para a extração de dados:

```
...
<main>
...
<table id="score">
  <thead>
    <tr>
      <th>Test</th>
      <th>Result</th>
    </tr>
  <tfoot>
    <tr>
      <th>Average</th>
      <td>82%</td>
    </tr>
  <tbody>
    <tr>
      <td>A</td>
      <td>87%</td>
    </tr>
    <tr>
      <td>B</td>
      <td>78%</td>
    </tr>
    <tr>
      <td>C</td>
      <td>81%</td>
    </tr>
</table>
...
</main>
```

Podemos realizar a extração tanto alcançando os elementos pelo método `getElementById()` quanto pelo API Selector. Usando o `getElementById()` teríamos:

```
var table = document.getElementById("score"),
    groups = table.tBodies,
    rows = null,
    cells = [];

for (var i = 0; i < groups.length; i++) {
  rows = groups[i].rows;
  for (var j = 0; j < rows.length; j++) {
    cells.push(rows[j].cells[1]);
  }
}

console.log(cells[0].innerText); // 87%
```

Esta abordagem tem as seguintes limitações:

1. Ela exige um elemento com atributo `id` como ponto de partida.
2. Foi necessário um esforço de programação para navegar pelos nós.
3. A estratégia de acesso é baseada em funções.

Como será demonstrado a seguir, a extração fazendo-se uso da API Selector torna o processo mais conciso e simples de implementar e interpretar. Os nós alvo são expressos por uma sentença:

```
var qs = document.querySelectorAll("table:nth-of-
type(1)>tbody>tr>td:nth-of-type(2)");

console.log(qs[0].innerText); // 87%
```

Desde o CSS3, muitos seletores foram adicionados como Seletores de Nível 3 (Kesteren, 2013), que agora podem ser usados não apenas nas folhas de estilo, como também no JavaScript a partir da API DOM.

Neste exemplo foi usado o método `querySelectorAll()` para retornar todos os nós que atendem à expressão, porém a Figura 4.5 também mostra um exemplo do método `querySelector()` que retorna um elemento alvo.

Um objeto Document dispõe de métodos para a criação dinâmica de elementos na árvore. Por exemplo, o método `createElement` cria um novo elemento. Cada nó possui um método `appendChild` para a inserção de novos nós como filhos. Do mesmo modo, `removeChild` remove tal associação de filho.

Na medida em que um aplicativo JavaScript interage com uma interface DOM modificando um documento HTML, o navegador atualiza dinamicamente a apresentação da página para refletir cada mudança. Isto permite tornar os documentos Web espaços de apresentação e interação para aplicativos JavaScript.

4.4.3. Eventos

Os eventos direcionam tudo que está acontecendo dentro do documento Web e está no núcleo do JavaScript, uma vez que ele está na origem. Tratando a questão de forma simplificada, um evento é um *broadcast* no documento DOM de cada acontecimento:

`click`, `scrollup`, `scrolldown`, `focus`, `blur` etc. Embora haja elementos que tenham eventos em comum, o modo como eventos são manipulados e o seu impacto serão diferentes.

Para ter uma visão de todos os eventos suportados pelo seu navegador, você pode executar a seguinte rotina no console do seu navegador (e.g., Firefox Firebug ou Chrome developer tools):

```
// Source:

// http://coding.smashingmagazine.com/2012/08/17/javascript-events-responding-user/
var log = function(what){ console.log(what) },
    i = '',
    out = [];
for (i in window) {
  if ( /^on/.test(i)) { out[out.length] = i; }
}
log(out.join(', '));
```

Considere o evento `click` e o exemplo de um "`form`" e um link "`a`" (âncora).

Se nós clicarmos em um "`input[type=submit]`" dentro do `form`, um novo evento é emitido pelo `form` e este evento irá ler o tag que é pai do `form`, e usar o valor do atributo chamado `method` para fazer uma chamada HTTP (seja GET ou POST) e submeter os valores do `form` baseado em cada valor de `input` disponível.

Um outro manipulador para o evento click é tag "`a`", quando ele recebe um clique. O manipulador executará uma ação equivalente a chamar o método `window.location`; ele lê o valor do atributo `a[href]` do elemento clicado e faz o navegador ler a nova localização.

Um detalhe importante a ser lembrado sobre manipulação e acionamento de eventos é que há muitos exemplos desatualizados em que é sugerido o uso de atributos do *tag* para disparar funções de evento, tal como o trecho de código a seguir. Esta prática era utilizada há muito tempo, mas a documentação a respeito não se atualiza rápido o suficiente.

```
// If you do this, please stop.
// HTML
<button onclick="javascript:sayhello()" >Hello!</button>

// JavaScript
function sayhello() {
  alert ('Hello!');
}
```

O problema desta técnica é que não há um controle central de eventos, que ficam espalhados em diversos pontos do documento. O que a documentação do Mozilla Developer Network e o jQuery sugerem é que você associe um manipulador de eventos a um nó via JavaScript e atribua uma função ao evento, como o exemplo a seguir:

```
// HTML
<button id="hello">Hello!</button>


// JavaScript
document.getElementById("hello").addEventListener("click",  function()
{
      alert('Hello!');
}, false);
```

4.5. Engenharia de Marcação e Estilos

4.5.1. Markup Patterns

A forma como usamos HTML, JavaScript e CSS para construir código reusável e eficiente tem relação direta com a velocidade de *renderização* dos nossos documentos.

Como produzir documentos HTML é trivial, as pessoas que estão construindo Web sites pela primeira vez são capturados nas mesmas "armadilhas".

Muitas práticas adotadas nas últimas décadas para o desenvolvimento de Web sites pode torná-los difíceis de manter. De fato, quantas vezes ouvimos dizer que alguém está apenas mudando o visual (*'skin'*) de uma aplicação preexistente?

A questão central torna-se então: por que não se usa o *markup* existente, mudando-se apenas o tema em um novo projeto? Não estamos considerando aqui aqueles casos em que o projeto tenha novos requisitos de software, uma nova estrutura, e nós queremos reimplementá-lo para torná-lo melhor, redefinindo inclusive as definições CSS.

O conhecimento dos conceitos arquiteturais por detrás do HTML, CSS e JavaScript é fundamental para potencializar a ação do desenvolvedor do *frontend*. Os casos comuns que tornam um documento difícil de manter são:

1. Redundância nos estilos, mesmo com o uso de propriedades herdadas.
2. Falta de uso de estilos herdados (quando apropriado), causando redundância.
3. Estilos demasiadamente específicos, quando partes dele poderiam ser reusadas.
4. Interpretação incorreta do *box model* atual.
5. Criação de estilos dependentes de localização.

Do mesmo modo que o domínio da arquitetura de software define seus *design patterns* (Gamma et al., 1995), é possível defini-los também no domínio do desenvolvimento de *frontends*. A seguir apresentamos patterns para a construção de CSS.

Para ajudar a descrever como criar um documento usando componentes reusáveis, Nicole Sullivan introduziu uma técnica chamada *Object Oriented* CSS - "OOCSS" (Sullivan, 2011).

Os princípios são simples e podem ser sumarizados como:

> *"Um 'Objeto' CSS é a repetição de um padrão visual, que pode ser abstraído em um fragmento independente de HTML, CSS e possivelmente JavaScript. Uma vez criado, um objeto pode ser então reusado ao longo do site." (Sullivan, 2011)*

Tal como os objetos, os estilos e sub-estilos (estilos componentes do estilo mais abrangente) devem ser concebidos hierarquicamente. Há muitos autores que documentam e nomeiam suas estratégias de atuação, mas todas elas têm uma coisa em comum: pense em um elemento visual como um componente e garanta que ele é autocontido.

Para que isto aconteça é necessário aplicar duas regras:

1. Separar a estrutura do visual (*skin*);
2. Separar o recipiente (*container*) do conteúdo.

Por exemplo, um elemento visual comum consiste em um conjunto de poucas palavras ao lado de uma imagem e, algumas vezes, alguns metadados associados. Vamos denominar este elemento de 'media', conforme exemplo ilustrado na Figura 4.11. Este elemento é muito comum e pode assumir várias formas. A Figura 4.11 mostra a imagem do lado esquerdo do texto, mas é possível, por exemplo, haver variantes com a imagem do lado direito.

Figura 4.11. Exemplo de uso do elemento media (fonte: http://www.stubbornella.org/).

O trecho de código a seguir ilustra a representação HTML + CSS do exemplo da Figura 4.11. Note que há uma concepção hierárquica baseada em objetos, em que os estilos `media-img` e `media-body` foram concebidos para o conteúdo que estará subordinado ao container `media`. Para tratar as variantes do estilo, a metodologia recomenda que a classe pai determine como deve se comportar os subordinados.

```
<div class="media attribution">
  <a href="http://twitter.com/stubbornella"
     class="media-img img">
    <img src="http://stubbornella.com/profile_image.jpg"
         alt="me" />
  </a>
  <div class="media-body">
    @Stubbornella 14 minutes ago
  </div>
</div>
```

```
/* ====== media ====== */
.media {margin:10px;}
.media, .media-body {
  overflow:hidden; _overflow:visible; zoom:1;}
.media .media-img {float:left; margin-right: 10px;}
.media .media-img img{display:block;}
```

O trecho ilustra o elemento media introduzido por Nicole Sullivan, mas alterado para refletir a metodologia SMACSS de Jonathan Snook (Snook, 2013).

Embora as metodologias sejam similares, elas não têm suas especificidades de acordo com o foco. Por exemplo, o OOCSS tem foco em construir coisas tão pequenas e modulares quanto possível, enquanto o SMACSS recomenda tornar-se explícito ao custo de se adicionar mais nomes de classes.

4.5.2. Criando um Módulo de Marcação

Tudo é uma caixa (“box”).

Uma vez construído, torna-se simples compreender a aplicação do elemento `media`, mas como podemos criar um novo elemento é outra questão. Para atender a esta demanda, Yandex criou uma metodologia que orienta o modo de fazer componentes (Yandex, 2013).

Ao invés de se criar componentes altamente especializados, é recomendado adotar uma visão diferente do problema e abstraí-lo que nos permita reusá-lo. Os tópicos a seguir sintetiza alguns dos aspectos chave da metodologia BEM (Yandex, 2013):

1. cada elemento visual é uma caixa;
2. caixas têm partes separadas;
3. caixas podem ter variantes;
4. partes e caixas têm uma representação básica e comportamento;
5. partes e caixas podem ter uma camada temática aplicada a eles.

O projeto de uma caixa inclui seu comportamento em JavaScript. Neste sentido, deve-se considerar o que pode acontecer com ela – incluindo eventos e mudanças de estado – e quais os impactos na caixa e na estrutura em que está inserida.

4.5.3. Pré-processadores CSS

Se por um lado o JavaScript está preparado para o reúso de patterns, uma estratégia equivalente para reuso de CSS ainda é um problema em aberto.

Embora o W3C e empresas que desenvolvem navegadores estejam empenhados na definição de componentes reusáveis, há algumas especificações que irão potencializar o modo como criamos marcações, dentre os recentes: Web Components, CSS Variables e CSS Fragmentation. Enquanto isso, há a necessidade de se criar componentes modulares com o que temos hoje.

Na criação de componentes, torna-se necessária a possibilidade de se criar variáveis, de se referir a outros *patterns* ("*mixins*") ou de se usar funções, e.g., para calcular variantes de cores. Para atender a esta demanda, surgiram os pré-processadores CSS. Um exemplo é o trecho CSS a seguir que usa variáveis, mixins e funções do LESS (Sellier, 2013):

```
@base: #f938ab;

.box-shadow(@style, @c) when (iscolor(@c)) {
  box-shadow:         @style @c;
  -webkit-box-shadow: @style @c;
  -moz-box-shadow:    @style @c;
}

.box-shadow(@style, @alpha: 50%) when (isnumber(@alpha)) {
  .box-shadow(@style, rgba(0, 0, 0, @alpha));
}

.box {
  color: saturate(@base, 5%);
  border-color: lighten(@base, 30%);
  div { .box-shadow(0 0 5px, 30%) }
}
```

Em LESS, variáveis e parâmetros são prefixados por `@`. No exemplo, a variável `@base` define o valor `#f938ab` a ser usado em vários pontos do CSS. O `.box-shadow` é um estilo reusável na forma de *pattern*. Ele recebe dois parâmetros que definem valores de algumas das suas propriedades. No exemplo, há duas possibilidades de `.box-shadow` com uma condição de escolha definida pela cláusula `when`. A classe `.box` instancia o `.box-shadow` dando valores aos seus parâmetros. Algumas funções usadas neste exemplo são: `iscolor()`, `isnumber()` e `lighten()`.

O LESS pode ser pré-processado no cliente ou no servidor. No caso do cliente, uma rotina em JavaScript carregada em tempo de execução realiza a conversão (Sellier, 2013):

```
<link rel="stylesheet/less" type="text/css"
href="styles.less" />

<script type="text/javascript">
    less = { env: "development" };
</script
<script src="less.js" type="text/javascript"></script>
```

Embora esta abordagem seja conveniente para uso em tempo de desenvolvimento e não exija um sistema de conversão rodando no servidor, ela não é adequada para distribuição, já que isto exigiria uma reconversão para cada visitante.

O pré-processamento no servidor distingue a versão local de desenvolvimento daquela para distribuição. A execução de um pré-processador *standalone* converte o CSS em LESS para uma versão estática com CSS puro.

Na medida em que o desenvolvimento envolve uma crescente quantidade de ferramentas, tais como as apresentadas nesta seção, há uma tendência recente em automatizar a sua ação coordenada e gerenciar o ciclo de vida de desenvolvimento das aplicações através de uma ferramenta de workflow chamada Yeoman (http://yeoman.io/).

4.5.4. Desenvolvimento JavaScript centrado em marcação

Nas seções anteriores foi destacada a importância de se estruturar *patterns* reusáveis para marcação. Nesta seção, estendemos o conceito para a sua aplicação no contexto de JavaScript, mais especificamente para uma técnica que chamaremos aqui de desenvolvimento centrado na marcação.

Um bom exemplo desta técnica é mostrado na documentação do *toolkit* Bootstrap (http://getbootstrap.com/2.3.2/javascript.html):

```
<div class="dropdown">
  <a class="dropdown-toggle" data-toggle="dropdown" href="#">
    Dropdown
  </a>
  <ul class="dropdown-menu" role="menu">
    ...
  </ul>
</div>
```

Este exemplo do `dropdown` ilustra o que é denominado data-api no Bootstrap, na qual é possível especificar e parametrizar um componente *dropdown* sem que se escreva uma única linha de código. Os componentes são parametrizados por metadados introduzidos na marcação e alguns dos componentes providos têm um comportamento JavaScript associado.

Componentes usando esta estratégia propiciam reuso e aumento de produtividade, já que eles cuidam de detalhes como o registro e manipulação de eventos, poupando esforço de codificação. Muitas interações podem ser generalizadas e

manipuladas de maneira similar. Um dos usos notáveis desta técnica foi no Facebook, quando eles otimizaram seu código JavaScript usando um 'Primer' (Adeagbo, 2013).

Este concepção de componente centrada nos dados será retomada na seção de Web Components.

O uso de `ids` (referenciando um único elemento) ao invés de classes (referenciando grupos de elementos) em estilos potencialmente reusáveis vai no sentido contrário das tendências aqui apresentadas e é considerada uma prática ruim. Estratégias de *markup*, como o Object Oriented CSS e o SMACSS, recomendam maximizar o reuso generalizando todos os padrões CSS, JavaScript e HTML.

4.6. Web Components

Web Components é uma iniciativa do WebApps WG de criar um modelo de componentes para a Web (Cooney & Glazkov, 2013).

O desafio de produzir um modelo de componentes para a Web (Web Component model) vai além de produzir um modelo de componentes de software para JavaScript. O núcleo da Web é construído sobre uma abordagem orientada a dados (data-driven), i.e. documentos são hospedeiros e dão estrutura às apps. A despeito da crescente relevância do JavaScript, o modelo centrado em documentos (document-centric ou data-centric) ainda prevalece.

Para tornar claro o que consideramos aqui um modelo de componentes data-centric, vamos partir de uma definição clássica de um componente, ilustrada na Figura Figura 4.12. Mesmo que não haja um consenso sobre a definição de componente de software (Broy et al., 1998), há uma distinção comum entre a interface pública – que apresenta uma abstração parametrizável do componente – e uma implementação privada – que define o comportamento ou serviço do componente. Os parâmetros da interface direcionam o seu comportamento. A interface tem dois papéis aqui: (i) ela abstrai os aspectos principais do componente, escondendo detalhes de implementação de um observador externo; (ii) ela torna explicitas as dependências/interações do componente com o sistema externo.

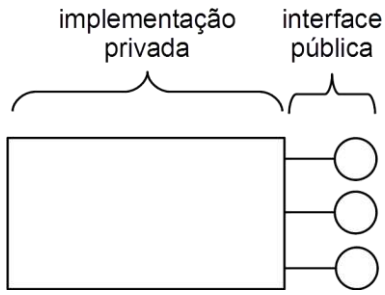


Figura 4.12. Diagrama de componente clássico.

4.6.1. *Templates*

Outro consenso relativo a componentes é que eles são blocos de conteúdo ou serviço reusáveis. Há diferentes perspectivas sobre o que pode ser este conteúdo, e.g., um código executável ou um binário. Em nosso contexto, o conteúdo será um segmento reusável de HTML/CSS/JavaScript.

Componentes precisam de um mecanismo independente para predefinir blocos de conteúdo antes do seu uso, de modo que fiquem disponíveis para serem usados tantas vezes quanto necessário. Este é o propósito dos *templates*.

O segmento a seguir ilustra como o carro SVG introduzido na Figura 4.4 pode ser transformado em um bloco reusável através de *templates*. Acima à esquerda aparece a descrição SVG do carro, apresentada anteriormente, na forma de *template*. Acima à direita é apresentada uma rotina em JavaScript responsável pela instanciação e uso *template*. Abaixo é apresentado o segmento no documento HTML referente ao ponto onde o template será inserido. A Figura 4.13 ilustra a estratégia de funcionamento dos templates em HTML.

```
<template
id="carComponent">

  <style scoped>
    …
  </style>

  <svg …>
    …
  </svg>


</template>
```

```
function applyTemplate()
{
  var carComponent =
                document.querySelector("#carComponent");

  var myCar = document.querySelector("#myCar");

  myCar.appendChild(carComponent.content.cloneNode());
}
```

```
<div id="myCar">
  <!-- empty -->
</div>
```

Conforme ilustrado no exemplo, cada template SVG, pode encapsular segmentos de um documento Web auto-contidos, ou seja, que representem uma sub-árvore DOM na íntegra. Isto inclui HTML, CSS, JavaScript e outras descrições XML aptas a serem incluídas em documentos HTML, tal como o SVG do exemplo. O CSS recebe um atributo “scoped” para restringir sua atuação no contexto do template.

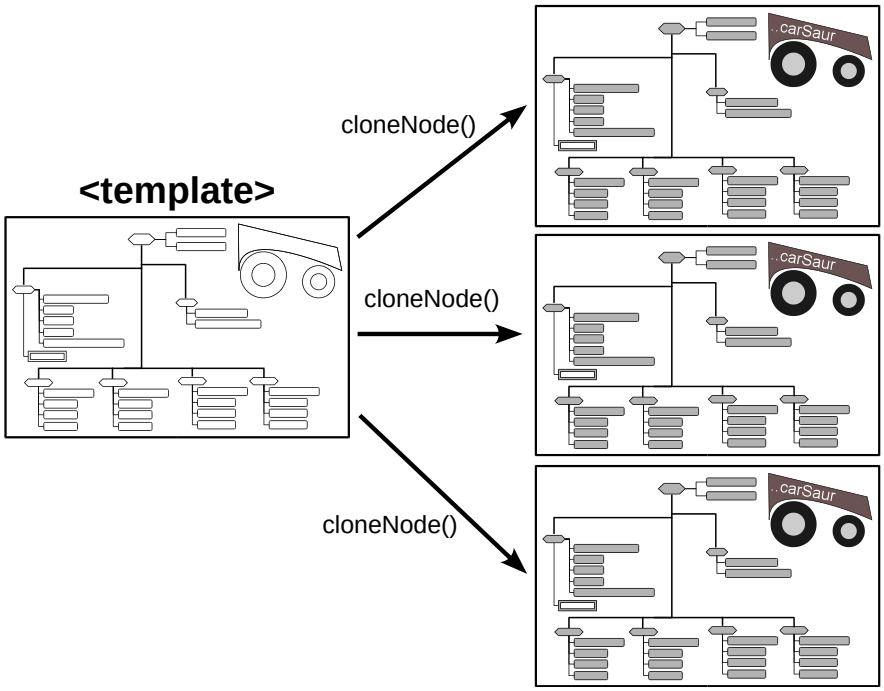


Figura 4.13. Instanciação de templates através da clonagem.

Como está ilustrado à esquerda da Figura 4.13, um *template* não é apresentado pelo navegador, mas seu conteúdo passa pelo parser e fica disponível para uso. O uso de um template é feito através de um método de clonagem do original, ativado pelo método `cloneNode()`.

Este processo está detalhado no método `applyTemplate()` apresentado anteriormente, que executa as seguintes tarefas:

1. Localiza o template dentro do documento a partir de seu identificador. Neste caso foi usada a função querySelector() que localiza um elemento a partir de uma query.
2. Localiza o alvo de inserção no documento Web. Trata-se da myCar, ilustrada abaixo do método.
3. Instancia o template a partir do processo de clonagem (função cloneNode()).
4. Adiciona a árvore instanciada no alvo.

Tão logo o template é clonado e inserido na árvore DOM ele passa a funcionar, e o carro aparece no documento HTML.

4.6.2. Shadow DOM

O tema encapsulamento em componentes de software pode se ampliar bastante quando aplicado ao contexto de Web. Uma das questões fundamentais diz respeito ao encapsulamento da árvore DOM. Ao se pensar em componentes Web que coexistem em um mesmo espaço de aplicação, o compartilhamento de uma grande árvore DOM pública e visível por todos pode se tornar um problema.

O shadow DOM (Cooney, 2013) (Glazkov, 2011) tem o propósito de encapsular árvores DOM na construção de componentes. Ele segue a lógica de encapsulamento em que parte da árvore DOM é pública (apenas a abstração do componente) e parte é privada (detalhes da sua implementação).

No texto anterior (Santanchè, 2013) apresentamos um exemplo do slider tratado por Glazkov (2011). Neste texto, apresentaremos a aplicação do conceito no componente carro. Considere que um desenvolvedor deseja criar um componente carro, de tal forma que ele possa configurá-lo facilmente através de marcação, como está ilustrado na Figura 4.14. No exemplo, a classe "`carComp`" associada ao marcador de id "`myCar`" indica que ele deseja instanciar um componente `car`. Para tornar o componente reusável, é importante que seja possível parametrizá-lo (e.g., cor, tamanho, tipo de roda, nome do carro etc.), de modo que o componente se adapte a diferentes contextos. Neste caso, foi escolhido um único parâmetro – o nome do carro – para fins de simplificação.

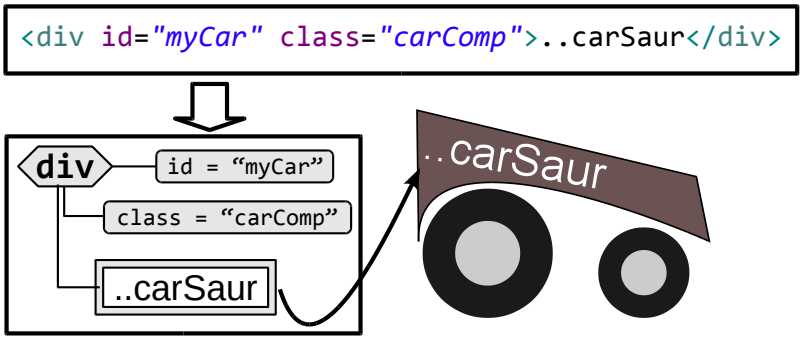


Figura 4.14. Interface de configuração de um componente carro.

Este parâmetro representa uma abstração externa simplificada. Ao alterar o nome do carro dentro da `<div>` de classe "`carComp`", o que de fato deve acontecer é que o nome do carro deve mudar no gráfico. Isto significa uma distinção entre a abstração pública e seu reflexo na implementação privada, como foi introduzido na Figura 4.12 e está ilustrado na Figura 4.15. A parametrização da abstração pública pode ser vista como uma árvore DOM simplificada, cujos valores afetarão uma segunda árvore DOM privada (shadow DOM), que esconde os detalhes SVG de como o carro é desenhado. Os valores modificados na interface devem refletir em pontos específicos da árvore privada que são chamados de *insertion points*.

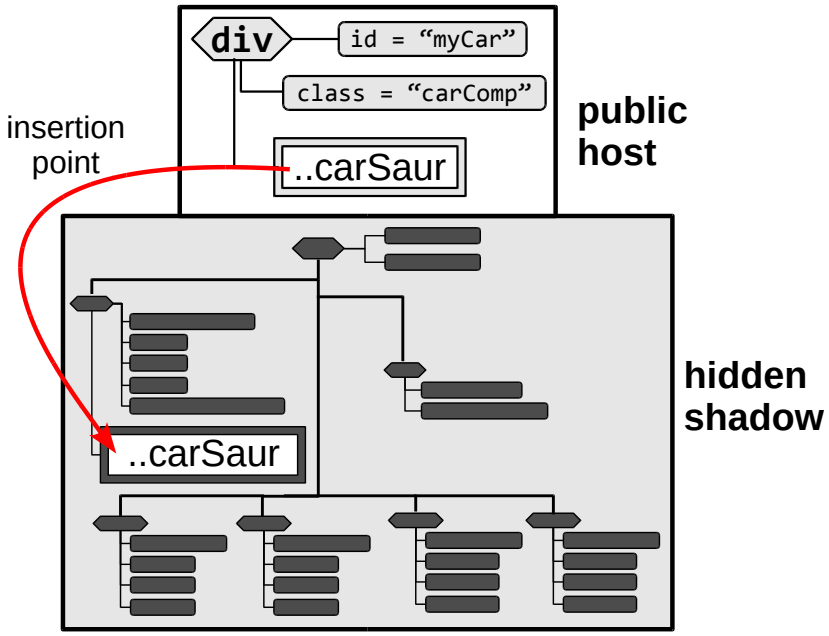


Figura 4.15. Divisão pública/privada do shadow DOM.

O shadow DOM possibilita que o aplicativo, ao acessar a árvore DOM, veja somente a parte pública ilustrada na Figura 4.14. A árvore que compõe esta abstração pública perceptível pelo JavaScript é chamada de host. Ela não é usada para renderizar a apresentação. Entretanto, é possível associar a esta árvore uma segunda responsável apenas pela apresentação. Esta árvore é conhecida como root. Ela não é visível pelo JavaScript mas será usada para montar a apresentação visual.

A árvore que ficará privada (shadow) pode ser definida na forma de template como ilustra a definição a seguir (esquerda). O ponto de inserção é definido através do tag `<content>`. Para fins de ilustração no exemplo, foi introduzido uma `<div>` externa à figura SVG onde será colocado o nome que está no parâmetro. Em testes realizados, o <contente> ainda não se comporta bem dentro do SVG, ao contrário do HTML. Uma estratégia para contornar este problema é fazer a substituição via JavaScript.

```
<template id="carComponent">

  <style scoped>
    …
  </style>

  <svg …>
    …
  </svg>

  <div id="textCarName"
       class="nameStyle"
       width="181px">
    <content></content>
  </div>
</template>
```

```
function applyTemplate()
{
  var myCar = document.querySelector("#myCar");

  var carComponent =
   document.querySelector("#carComponent").content;

  var myCarShadow = myCar.webkitCreateShadowRoot();
  // standard: createShadowRoot()

  myCarShadow.appendChild(carComponent);
}
```

```
<div id="myCar" class="carComp">..carSaur</div>
```

No código JavaScript são executadas as seguintes tarefas:

1. O host identificado por `myCar` é recuperado. Neste caso, foi usado o identificador para fins de simplificação mas, para fins de reúso, é mais adequada a recuperação a partir da classe.
2. É recuperado o root do *template* que será usado para a criação da parte escondida que descreve o carro em SVG.
3. É montada uma árvore shadow DOM do host através do método createShadowRoot() (este é o método padrão; no exemplo foi usada uma implementação provisória que roda no Google Chrome).
4. A árvore shadow é associada ao root do *template*. Neste momento os parâmetros do host irão automaticamente alimentar os pontos de inserção.

É possível haver mais de um parâmetro no host. Cada parâmetro estará em um atributo ou sub-elemento do host. O exemplo a seguir ilustra este caso, em que há um elemento identificado como "`carName`", dentro do elemento host, com o nome do carro. Para este caso, o ponto de inserção pode incluir um parâmetro `select` (vide exemplo a seguir) e selecionar o que irá recuperar. A expressão segue a abordagem da API Selector, descrita na Seção 4.4.

```
<div id="myCar" class="carComp">
  <div id="carName">..carSaur</div>
</div>
```

```
<content select="#carName"></content>
```

4.7. Evolução das arquiteturas de aplicativos móveis

4.7.1. Aplicativos usando tecnologias Web como tecnologia secundária

Em inúmeras plataformas, tecnologias Web não estavam na lista oficial de tecnologias suportadas para o desenvolvimento de aplicações, mas sim para papéis menores. Nessas plataformas, tecnologias Web só podem ser utilizadas pela inserção na aplicação de um componente nativo, o webview, que funciona como um motor Web para o qual são despachadas as partes Web. Dessa forma, essas plataformas suportam HTML, CSS e JavaScript como recursos complementares e via o componente webview. As aplicações não são desenvolvidas integralmente com as tecnologias Web por não serem suportadas direta/nativamente.

Por esta razão, para os casos em que se deseja criar um aplicativo usando integralmente tecnologias Web, é necessário utilizar um aplicativo-envoltório, conforme ilustrado na Figura 4.16, implementado em alguma das tecnologias nativas, que faz o bootstrap do motor e o aplicativo Web dentro dele. Para utilizar os recursos de hardware, são usadas bibliotecas híbridas, que de um lado se apresentam como uma API JavaScript para que o aplicativo Web tenha acesso ao hardware, e do outro são implementadas em código nativo para acessar o hardware usando as APIs nativas. Uma

das bibliotecas mais conhecidas que implementa essas duas partes é o Phonegap, da Adobe, e sua equivalente livre, Cordova, da Apache. A maior parte das plataformas utiliza essa abordagem, incluindo Android e iOS.

Figura 4.16. Web App com aplicativo-envoltório.

Outra alternativa está ilustrada na Figura 4.17. O código escrito em HTML, CSS e JavaScript é compilado para código nativo da plataforma específica em que ele será executado. Deste modo, o mesmo código pode ser compilado para diversas plataformas. Exemplos desta tecnologia são o Titanium Mobile Development Environment (http://www.appcelerator.com/platform/titanium-platform/) e o Marmelade (http://www.madewithmarmalade.com/).

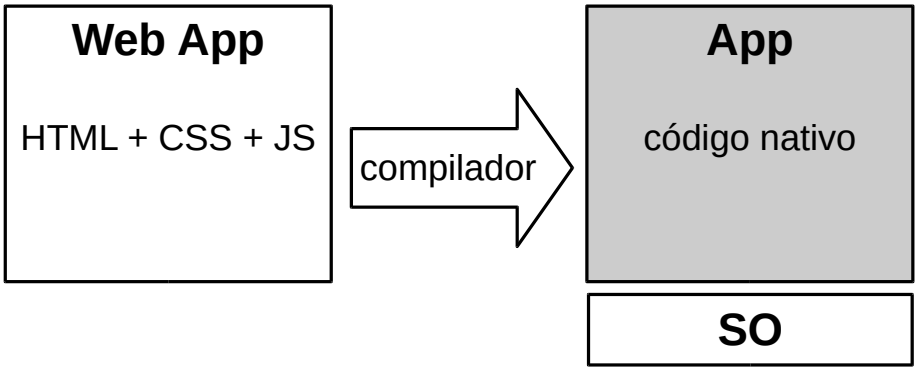


Figura 4.17. Web App compilada para a plataforma.

4.7.2. Aplicativos usando tecnologias Web como tecnologia primária

As plataformas que não oferecem tecnologias Web por padrão – apresentadas na seção anterior – possuem muitos problemas. Entre eles, os principais são o acesso ao hardware, que depende de bibliotecas de terceiros e não é uniforme, e o desempenho, que tende a ser muito menor que na plataforma nativa por conta da quantidade de indireções.

Esta seção analisará como algumas plataformas tentaram resolver isso em duas diferentes estratégias: (i) como não havia especificação padrão Web para acessar os componentes de hardware diretamente, cada plataforma criou uma API Web própria para acesso a tais componentes; (ii) a plataforma oferece uma solução híbrida combinando desenvolvimento nativo e Web, que dispõe de apenas parte das ferramentas Web para o desenvolvimento de aplicações. No caso (i), tem-se uma abordagem muito parecida com a anterior, mas nativa, e não dependente de uma biblioteca externa,

implementada por terceiros. No caso (ii), tem-se um desenvolvimento mais próximo do nativo, com uma curva de aprendizado menor, em que parte do conhecimento com tecnologias Web pode ser reaproveitado. Em ambas as soluções os problemas de desempenho também são diminuídos, já que o suporte é nativo, como ilustra a Figura 4.18.

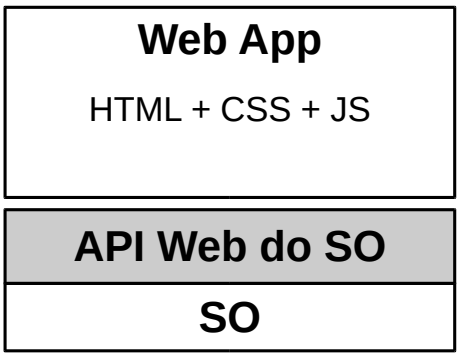


Figura 4.18. Web App com API suportada pelo SO.

Várias plataformas mais recentes utilizam essa abordagem. Alguns exemplos são BlabkBerry 10 – abordagem (i) – e Windows 8 – abordagem (ii).

4.7.3. Aplicativos Web como aplicativos nativos ou de primeira classe

Nesse caso, tecnologias Web padrão compõem o mecanismo nativo para construção de aplicações. Por nativo entende-se que todo o processo de interação com o SO e o hardware – incluindo-se dispositivos de apresentação – é feito por meio de protocolos e padrões Web. A Figura 4.19 ilustra esta abordagem de aplicativo, que pode ser confrontada com aquela ilustrada na Figura 4.16.

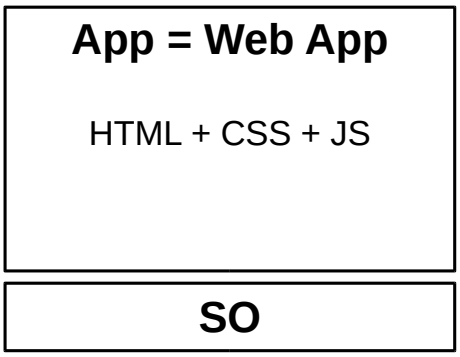


Figura 4.19. Web App nativa.

Fazendo-se um paralelo: as plataformas Android e iOS possuem uma API própria para interação com o display, que pode ser acessada por qualquer linguagem de programação considerada nativa para a plataforma; plataformas Web nativas, por outro lado, usam apenas as APIs Web padrão, definidas em conjunto com orgãos como o W3C e WHATWG. No segundo caso, a disposição visual dos elementos, por exemplo, fica a cargo do HTML+CSS, que é manipulado via JavaScript, da mesma forma que o seria em qualquer navegador Web. Adicionalmente, aplicações para plataformas desse tipo também funcionam (ou deveriam funcionar, dados padrões que ainda estão em

construção) em qualquer navegador, independendo de plataforma ou sistema operacional, dependendo exclusivamente do suporte dessa plataforma aos padrões Web.

Essa proposta é a mais alinhada com aquela aberta e livre da Web. Entretanto, ainda há limitações a serem superadas, já que não existem especificações padronizadas para as APIs que dão acesso ao componentes de hardware. Portanto, plataformas que querem seguir nesta linha devem colaborar para construir novos padrões. O Firefox OS é um exemplo de plataforma que utiliza essa abordagem, ainda muito pouco explorada.

4.8. Planejamento para criação de interfaces e considerações sobre usabilidade

Todo sistema homem-máquina tem como objetivo facilitar à execução de tarefas a serem realizadas por humanos. Por isso, considerar o usuário como foco principal ao desenvolver o sistema homem-máquina é fator determinante como critério de sucesso do sistema.

Existem diversas métodos que visam otimizar à concepção de um sistema. Por exemplo, há o ciclo de desenvolvimento em V e em cascata. Mas, entre estes métodos, a norma ISO 9241-210 (2010) é a que tem maior foco no usuário e deve ser aplicada ainda antes da fase de concepção.

A norma ISO 9241-210 é dividida em etapas: Planificar o processo de concepção, definir o uso de contexto e suas limitações, definir as especificações das exigências (*must*, *should*, *could*), conceber soluções e, por fim, avaliar o produto de acordo com as exigências. Cada etapa será detalhada nos parágrafos a seguir.

4.8.1. Planificar o processo de concepção

A primeira etapa da norma ISO 9241-210, planificar o processo de concepção, consiste em definir o tamanho da equipe que irá participar do projeto de desenvolvimento do sistema, bem como a definição do cronograma e do orçamento. A concepção deve ser baseada sobre uma compreensão clara de quem são os usuários, as tarefas e ambiente de uso do sistema. Os usuários devem participar durante todo o processo de desenvolvimento do sistema e as equipes de concepção devem pertencer a equipe multidisciplinares.

4.8.2. Especificar o contexto de uso

Para especificar o contexto de uso, deve-se primeiro, definir os seguintes aspectos:

- Quem são os usuários primários e secundários.
- Quais os objetivos e as tarefas a serem executadas.
- Qual o ambiente/contexto de uso.

4.8.3. Especificar as necessidades dos usuários

Nesta etapa, deve-se reunir todas as informações que foram levantadas e traduzi-las em requisitos e exigências funcionais, que devem refletir os objetivos do sistema e dos usuários para com o sistema.

Deve-se também, considerar todas as limitações do sistema. Desta forma, no momento da decomposição das tarefas e dos objetivos relacionados à eficácia, esses podem ser considerados de forma optimizada.

De acordo com a norma ISO 9241-11, a eficácia, eficiência e a satisfação são definidos como:

- Eficácia: precisão e completitude com que os usuários alcançam os objetivos.
- Eficiência: os recursos utilizados em relação à exatidão e exaustão que os usuários alcançam seus objetivos.
- Satisfação: quando não há desconforto e há atitudes positivas por parte dos usuários ao utilizar o sistema.

4.8.4. Produzir soluções de concepção que correspondem as necessidades dos usuários

Uma forma de verificar se a concepção corresponde às exigências dos usuários, é desenvolver protótipos com diferentes níveis de fidelidade. Inicialmente, a criação de uma versão de baixa fidelidade e, posteriormente, adicionar mais detalhes de forma que os membros da equipe de concepção possam validar os requisitos.

Os protótipos considerados de baixa fidelidade (no caso de interfaces são conhecidos como wireframes e sketches) são os que oferecem um mínimo para ilustrar um caso e é útil como apoio para explicar uma ideia. Eles podem ser fabricados de diversos tipos de materiais, por exemplo, em papel, em cartolina etc. Eles são geralmente simples, rápidos e baratos a serem produzidos. A principal vantagem deste tipo de protótipo, é que ele pode ser facilmente modificado para ajudar na exploração de ideias e concepções alternativas.

O último nível de prototipagem deve ser o mais próximo do produto final. É interessante desenvolver um protótipo do sistema o mais real possível para que usuários possam testá-lo. Esse tipo de experimentação tem um grande valor por validar os conceitos antes de colocá-los em produção final.

4.8.5. Verificação e validação da interface

Depois de conceber os diferentes tipos de soluções, é importante verificar o sistema em relação às exigências do projeto, bem como, entender como as limitações do sistema podem afetar os usuários. Desta forma, a concepção será guiada por avaliações detalhadas e centradas no usuário. Este processo interativo garantirá que cada fase ajude a melhorar o caso de uso e aumentará a qualidade do produto a cada protótipo criado.

4.8.6. Avaliar a concepção de acordo as exigências

Uma vez que o sistema é desenvolvido, existem dois métodos que podem ser considerados para verificar se todas as condições do sistema foram atendidas:

- O uso da lista de verificação em conformidade da ISO.
- Os testes de usabilidade para fim de validação.

4.8.7. Lista de verificação em conformidade da ISO 9241-210

Uma lista de elementos deve ser usada para validar e assegurar que nenhum ponto foi esquecido em relação à norma em vigor.

Uma lista permite, entre outros:

- determinar quais as recomendações são aplicáveis;
- determinar se as recomendações aplicáveis foram seguidas;
- fornecer uma lista de todas as recomendações aplicáveis e se elas foram seguidas.

Independente do método de verificação considerado o mais adequado, a lista proposta pela ISO 9241-151 oferece um espaço para indicar o nível de conformidade, bem como, os métodos utilizados que podem ser inscritos na coluna de comentários.

Os números e os títulos dos Artigos/Parágrafos estão presentes dentro das duas primeiras colunas da tabela. A terceira coluna é utilizada para indicar se a recomendação dentro de cada artigo ou parágrafo é aplicável ou não. Todos esses aspectos devem ser verificados em que se concerne ao quadro de concepção do sistema.

4.8.8. Analise das tarefas com os usuários - Validação

O teste de validação é efetuado em uma etapa avançada do processo para verificar a usabilidade do sistema. O objetivo é controlar a conformidade da interface do sistema. As avaliações de teste de usabilidade de sistemas interativos são geralmente constituídas de um conjunto de tarefas que os usuários devem executar a fim de resolver problemas.

Com as tarefas apropriadas para avaliação, pode-se mensurar a facilidade de utilização através de variáveis objetivas e subjetivas.

De acordo com a norma ISO 9241-11, depois da identificação dos objetivos, deve-se determinar os resultados desejáveis, as zonas aceitáveis para definir um nível de satisfação. Existem cinco parâmetros que devem ser considerados para avaliar a usabilidade:

- Fácil de aprender, o usuário pode interagir rapidamente com o sistema, o aprendizado das funcionalidades e dos botões de navegação.
- Eficaz de usar, uma vez que o usuário compreende como o sistema funciona, ele é capaz de encontrar a informação que ele necessita.
- Fácil de reter, mesmo por um usuário que não utiliza o sistema diariamente, ele é capaz de lembrar as principais funcionalidades do sistema.
- Poucas chances de erro, os usuários não comentem muitos erros e são igualmente capazes de corrigir um erro.
- Agradável de usar, os usuários se sentem satisfeitos com o sistema, da forma que eles interagem.

4.9. Considerações Finais

Este texto apresentou uma panorâmica de aspectos relevantes do desenvolvimento de aplicativos independentes de plataforma para dispositivos móveis, baseado em tecnologias Web. Ele tratou de diversos projetos e padrões que ainda estão em formação, tentando capturar os elementos fundamentais de cada um deles.

O Javacript já foi duramente criticado nos últimos anos em razão de problemas de desempenho, devido à sua natureza interpretada e dificuldades na fase de desenvolvimento para a equipe de programação. Atualmente, com a chegada dos navegadores modernos e o avanço das plataformas móveis, observou-se uma grande melhora no tempo de execução, em razão de otimizações, compilação e outras técnicas utilizadas.

Particularmente no desenvolvimento de aplicativos de alto desempenho como jogos, uma das mais importantes variáveis dependentes do desempenho é conhecida como FPS (*Frames per Second*). Como jogos exibem animações, o controle de tempo é necessário para suavizar as animações apresentadas, por isso o controle do tempo (ou de quantos quadros é possível desenhar por segundo) é muito importante para o sucesso de uma implementação. O HTML5 trouxe, portanto, um método – que ainda está em fase de uniformização entre os navegadores – que permite ao desenvolvedor determinar tarefas a serem executadas, toda vez que o navegador está pronto para atualizar seu conteúdo, ou seja, o navegador toma a responsabilidade pelo controle de FPS.

Ainda por se tratar de uma linguagem interpretada, o JavaScript apresenta alguns problemas de segurança, uma vez que permite ataques de *injection*, que podem alterar completamente o funcionamento do aplicativo, abrindo brechas para invasão em aplicações que estão sendo executadas na plataforma do usuário final, dificultando portanto a venda de soluções nesta plataforma.

Outro ponto importante acerca do JavaScript está no fato de que, apesar de ser uma especificação padronizada, a implementação desta é dependente de cada navegador, e os desenvolvedores vivenciam dificuldades diversas. Isso torna difícil a tarefa de criar um código único e 100% funcional independente de plataforma. Adicionalmente, observam-se inconsistências no resultado produzido entre diferentes navegadores, ou seja, dois navegadores diferentes executam o mesmo código, porém o efeito observado diverge de alguma forma.

Este problema motivou o desenvolvimento de diversos frameworks que visam homogeneizar variações entre navegadores e simplificar o trabalho de equipes de programação, uma vez que oferecem artefatos – por exemplo, classes e métodos – que resolvem problemas recorrentes para desenvolvedores com essas tecnologias.

Agradecimentos

Este trabalho foi parcialmente financiado pela FAPESP, o Microsoft Research FAPESP Virtual Institute (projeto NavScales), CNPq (projeto MuZOO) e PRONEX-FAPESP, INCT in Web Science (CNPq 557.128/2009-9) e CAPES, bem como financiamento individual do CNPq.

Ilustrações de cabeças de dinossauros, usadas nas figuras, cedidas sob licença Creative Commons Attribution-Noncommercial-No Derivative por *highdarktemplar (http://highdarktemplar.deviantart.com/).

Referências

Adeagbo, M. (2013). Primer. Retrieved from http://youtu.be/wHlyLEPtL9o.

Adida, B., & Birbeck, M. (2008). RDFa Primer. online: www.w3.org/TR/2008/NOTE-xhtml-rdfa-primer-20081014/

Adida, B., Birbeck, M., & Pemberton, S. (2011). HTML+RDFa 1.1 -- W3C Working Draft 13 January 2011. online: http://www.w3.org/TR/2011/WD-rdfa-in-html-20110113/

Atkins Jr., T., Etemad, E. J., Mogilevsky, A., Baron, L. D., Deakin, N., Hickson, I., & Hyatt, D. (2012). CSS Flexible Box Layout Module - W3C Candidate Recommendation, 18 September 2012. online: http://www.w3.org/TR/css3-flexbox/

Berjon, R., Leithead, T., Navara, E. D., O'Connor, E., Pfeiffer, S., & Hickson, I. (2012). A vocabulary and associated APIs for HTML and XHTML - W3C Candidate Recommendation 17 December 2012. online: http://www.w3.org/TR/2012/CR-html5-20121217/

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., & Yergeau, F. (2008). Extensible Markup Language (XML) 1.0 (Fifth Edition) -- W3C Recommendation 26 November 2008.

Broy, M., Deimel, A., Henn, J., Koskimies, K., Plášil, F., Pomberger, G., Pree, W., Stal, M., Szyperski, C. (1998). What characterizes a (software) component? Software -- Concepts & Tools, 19(1), 49–56.

Cooney, D., & Glazkov, D. (2013). Introduction to Web Components - W3C Editor's Draft 13 April 2013. online: https://dvcs.w3.org/hg/webcomponents/raw-file/tip/explainer/index.html

Cooney, D. (2013). Shadow DOM 101. HTML5 ROCKS. online: http://www.html5rocks.com/en/tutorials/webcomponents/shadowdom/

Dahlström, E., Dengler, P., Grasso, A., Lilley, C., McCormack, C., Schepers, D., Watt, J., et al. (2011). Scalable Vector Graphics (SVG) 1.1 (Second Edition) - W3C Recommendation 16 August 2011. online: http://www.w3.org/TR/2011/REC-SVG11-20110816/

Ecma International (2011). ECMAScript Language Specification - Standard ECMA-262 (5.1 ed.).

Fowler, M. (2012). Developing Software for Multiple Mobile Devices. online: http://martinfowler.com/articles/multiMobile/

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design patterns: elements of reusable object-oriented software. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Glazkov, D. (2011). What the Heck is Shadow DOM? online: http://glazkov.com/2011/01/14/what-the-heck-is-shadow-dom/

Hickson, I. (2011). HTML Microdata -- W3C Working Draft 13 January 2011. W3C. Retrieved from http://www.w3.org/TR/2011/WD-microdata-20110113/

ISO 9241-210 (2010). Ergonomics of human-system interaction -- Part 210: Human-centered design for interactive systems. (Geneva: International Standards Organization).

ISO 9241-151 (2008). Ergonomics of human-system interaction -- Part 151: Guidance on World Wide Web user interfaces. (Geneva: International Standards Organization).

Kelly, H. (2013). "Facebook mobile users surpass desktop users for first time". CNN. Online: http://edition.cnn.com/2013/01/30/tech/social-media/facebook-mobile-users

Kesteren, A. van, Gregor, A., Hunt, L., & Ms2ger. (2012). DOM4 - W3C Working Draft 6 December 2012. online: http://www.w3.org/TR/2012/WD-dom-20121206/

Kesteren, A. van, & Hunt, L. (2013). Selectors API Level 1 - W3C Recommendation 21 February 2013. Retrieved from http://www.w3.org/TR/2013/REC-selectors-api-20130221/

Khare, R. (2006). Microformats: The Next (Small) Thing on the Semantic Web - IEEE Internet Computing, 10(1), 68–75.

Lie, H. W., Çelik, T., Glazman, D., & Kesteren, A. van. (2012). Media Queries - W3C Recommendation 19 June 2012.

Manola, F., & Miller, E. (2004). RDF Primer -- W3C Recommendation. Retrieved from http://www.w3.org/TR/2004/REC-rdf-primer-20040210/

Marcotte, E. (2010, May 25). Responsive Web Design. online: http://alistapart.com/article/responsive-web-design.

Santanchè, A., & Panaggio, R. (2013). Desenvolvimento Independente de Plataforma para Dispositivos Móveis usando Tecnologias Web. In Atualizações em Informática 2013 (pp. 74–139). Maceió: Edufal.

Snook, Jonathan. (2013). SMACSS. . Retrieved August 27, 2013, from http://smacss.com/

Sellier, A. (2013). Leaner CSS ("LESS"). Retrieved August 27, 2013, from http://lesscss.org/

Sullivan, N. (2011). Object Oriented CSS. Retrieved August 26, 2013, from https://github.com/stubbornella/oocss/wiki

W3C SPARQL Working Group (2013). SPARQL 1.1 Overview - W3C Recommendation 21 March 2013. Retrieved from http://www.w3.org/TR/2013/REC-sparql11-overview-20130321/

Yandex. (2013). BEM. Retrieved August 27, 2013, from http://bem.info/