

LIGA DE ENSINO DO RIO GRANDE DO NORTE
FACULDADE NATALENSE PARA O DESENVOLVIMENTO DO RIO GRANDE DO
NORTE
CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

Karlisson de Macedo Bezerra

INK2CANVAS: UMA EXTENSÃO DO INKSCAPE PARA CONVERTER SVG EM
HTML5 CANVAS

NATAL/RN
2011

KARLISSON DE MACEDO BEZERRA

**INK2CANVAS: UMA EXTENSÃO DO INKSCAPE PARA CONVERTER SVG EM
HTML5 CANVAS**

Monografia apresentada ao curso de Bacharelado em Sistemas de Informação da Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte como requisito parcial para obtenção do grau de Bacharel em Sistemas de Informação.

Orientador(a):

Joseane Pinheiro Alves, Me.

NATAL/RN

2011

Catálogo na Publicação – Biblioteca da FARN
Setor de Processos Técnicos

Bezerra, Karlisson de Macedo

Ink2canvas: uma extensão do Inkscape para converter SVG em HTML5
Canvas / Karlisson de Macedo Bezerra. – Natal, 2011.
52f.

Orientador: Joseane Pinheiro Alves (Msc.)

Monografia (Bacharelado em Sistemas de Informação) – Faculdade
Natalense para o Desenvolvimento do Rio Grande do Norte.

1. Extensão - Monografia. 2. SVG – Monografia. 3. HTML – Monografia.
4. Canvas – Monografia. 5. Inkscape – Monografia. I. Título.

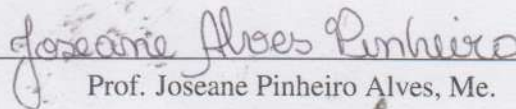
RN/FARN/BC

CDU 004

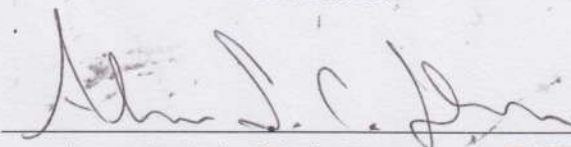
KARLISSON DE MACEDO BEZERRA

**INK2CANVAS: UMA EXTENSÃO DO INKSCAPE PARA CONVERTER SVG EM
HTML5 CANVAS**

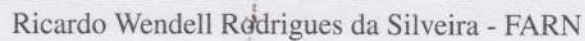
Monografia apresentada e aprovada em _____ de _____ de _____, pela
banca examinadora composta pelos seguintes membros:



Prof. Joseane Pinheiro Alves, Me.
Orientadora



Alexandre Luiz Galvão Damasceno - FARN



Ricardo Wendell Rodrigues da Silveira - FARN

"A stupid man's report of what a clever man says can never be accurate, because he unconsciously translates what he hears into something he can understand."

– Bertrand Russell.

RESUMO

O objetivo deste trabalho é desenvolver uma extensão para o *software* de ilustração vetorial *Inkscape*, com o propósito de converter seu formato de arquivo padrão, SVG, para o HTML5 *Canvas*, com base no estudo prévio desses dois padrões *web*. Os resultados desse estudo serão úteis para a compreensão dos padrões gráficos abertos e a criação de aplicações *web* gráficas e interativas. Mediante pesquisa bibliográfica para analisar e comparar os dois padrões, foram obtidas informações relevantes para a construção da extensão. Após ser desenvolvida, a extensão passou por várias melhorias, assim como por análises de performance para determinar seus pontos fracos. As características observadas no estudo dos padrões, em conjunto com os resultados do software desenvolvido, servem de apoio para a comparação dos padrões e o uso mais indicado de cada tecnologia, que dependerá do tipo de aplicação gráfica a ser desenvolvida. Finalmente, foram realizados vários testes que permitiram verificar a eficácia da extensão através da análise das imagens e códigos gerados.

Palavras-chave: Extensão. SVG. HTML. Canvas. Inkscape

ABSTRACT

This work's goal is to develop an extension for the vector illustration software Inkscape, in order to convert its default file format, SVG, to HTML5 Canvas, based on previous study of these two web standards. The results of this study will be useful for the comprehension of open graphic standards and development of graphic and interactive web applications. Based on a bibliographic research to analyze and compare the two standards, relevant information was obtained for the extension development. After being developed, many improvements has been made to the extension, as well as a performance analysis to determine its weaknesses. The features observed in the study of standards, together with the results of the developed software, will be useful for the understanding and comparison of the standards and the most appropriate use of each technology, which will depend on the type of graphics application being developed. Finally, several tests were performed to verify the effectiveness of the extension by comparing the images and codes generated.

Keywords: Extension. SVG. HTML. Canvas. Inkscape

SUMÁRIO

1	INTRODUÇÃO	9
2	OBJETIVOS	10
2.1	OBJETIVO GERAL	10
2.2	OBJETIVOS ESPECÍFICOS	10
3	JUSTIFICATIVA	11
4	REFERENCIAL TEÓRICO	12
4.1	HTML5 CANVAS	12
4.1.1	Histórico	12
4.1.2	Introdução ao Canvas	13
4.1.3	O contexto de renderização	15
4.1.4	Formas, caminhos e curvas	17
4.1.5	Atributos gráficos	20
4.1.6	Transformações	24
4.1.7	Salvando e restaurando o contexto	26
4.1.8	Área de clip	26
4.1.9	Texto	27
4.1.10	Imagens	29
4.2	SVG	30
4.2.1	Definição	30
4.2.2	Histórico	31
4.2.3	Sistemas gráficos	31

4.2.4	Formas básicas	32
4.2.5	Caminhos livres	33
4.2.6	Texto	34
4.2.7	Estilo	35
4.2.8	Transformações	36
4.2.9	Grupos	36
4.2.10	Referências e definições	36
4.2.11	Clips	37
4.3	O INKSCAPE E A ESTRUTURA DE EXTENSÕES	38
4.3.1	Definição e histórico	38
4.3.2	Extensões	39
4.4	TRABALHOS RELACIONADOS	41
4.4.1	canvg	41
4.4.2	From SVG to Canvas and Back	41
5	METODOLOGIA	43
6	ANÁLISE DE RESULTADOS	44
6.1	SVG E HTML5	44
6.2	O INKSCAPE E O SVG	46
6.3	DESENVOLVIMENTO E PROCESSO DE CONVERSÃO	46
6.4	TESTES REALIZADOS	47
7	CONCLUSÃO	50
7.1	TRABALHOS FUTUROS	51
	REFERÊNCIAS	52

LISTA DE FIGURAS

Figura 1 - Representação do sistema de coordenadas do <i>Canvas</i>	16
Figura 2 - Exemplo de retângulos em <i>Canvas</i>	18
Figura 3 - Exemplos de caminhos	20
Figura 4 - Exemplo de gradiente linear	21
Figura 5 - Exemplo de gradiente radial	21
Figura 6 - Exemplo de atributos de linha	22
Figura 7 - Sombras	23
Figura 8 - Tipos de atributos de composição	24
Figura 9 - Exemplo de transformações no <i>Canvas</i>	25
Figura 10 - Exemplo de <i>clip</i>	26
Figura 11 - Esquema da área de <i>clip</i>	27
Figura 12 - Atributos de texto	28
Figura 13 - <i>Sprites</i> utilizados em jogos	30
Figura 14 - Representação de sistemas gráficos	32
Figura 15 - Tela principal do <i>Inkscape</i>	39
Figura 16 - Teste de conversão com elementos básicos	48
Figura 17 - Teste de área de <i>clip</i> e transformações	48
Figura 18 - Teste com desenhos mais complexos	49
Figura 19 - Teste com textos	49

1 INTRODUÇÃO

A *Internet* deu início a uma das grandes revoluções tecnológicas da humanidade. Essa fantástica invenção aproximou pessoas, criou novas formas de comunicação e interação social, eliminou barreiras geográficas e criou novas perspectivas econômicas. A criação e consequente evolução da *web* se deu através do uso de padrões e protocolos abertos que possibilitaram o desenvolvimento de inúmeras aplicações e tecnologias. Essa diversidade é decorrente da abrangência e facilidade de adoção dos padrões. Nesse contexto, novas ideias e aplicações surgem continuamente, e novos padrões são criados e atualizados para atender às necessidades dessa plataforma em constante mudança.

Dentre esses novos padrões destacam-se o SVG e o HTML5, acrescentando vários recursos gráficos que enriquecem a experiência do usuário, bem como facilitam o desenvolvimento de aplicações *web* interativas. A compreensão dos padrões gráficos abertos, assim como de suas vantagens e pontos fracos, é de fundamental importância para estabelecer boas práticas e auxiliar na escolha das tecnologias e recursos adequados, dado que apresentam diferenças conceituais. Um dos aspectos importantes da *web* reside na transparência, interoperabilidade e portabilidade de seus padrões. É imprescindível que os dados possam ser visualizados em diferentes formatos e dispositivos, devido às necessidades casuais e à própria característica de universalidade da *web*. Por isso, são necessárias ferramentas para auxiliar nesse processo.

A proposta desse trabalho é contribuir com a evolução da *web* desenvolvendo uma ferramenta de conversão do SVG para o HTML5 mediante estudo comparativo entre esses padrões, estimular a sua adoção e a implementação de projetos de *software* semelhantes.

2 OBJETIVOS

2.1 OBJETIVO GERAL

Desenvolver uma ferramenta de conversão do formato SVG para o HTML5 que funcione como uma extensão do *software* de ilustração vetorial *Inkscape*.

2.2 OBJETIVOS ESPECÍFICOS

- Analisar o padrão HTML5 em relação ao suporte à criação de gráficos com o elemento *Canvas*, apresentando exemplos de sua utilização.
- Analisar o padrão gráfico vetorial SVG e seus elementos básicos, bem como sua implementação pelo *Inkscape*.
- Realizar um estudo comparativo entre o SVG e o HTML5, definindo em quais casos cada tecnologia é melhor aplicada, apresentando pontos positivos e negativos.
- Apresentar testes da extensão de conversão e utilizar os dados de saída para realizar uma análise dos resultados.
- Exemplificar casos de uso para a extensão, além de definir metas de desenvolvimento para garantir a continuidade e melhoria do projeto.

3 JUSTIFICATIVA

O desenvolvimento de sites interativos e aplicações *web* tem sido facilitado a cada nova tecnologia incorporada ao conjunto de ferramentas e padrões normalmente utilizados. O recente padrão HTML5 permite a criação de aplicações *web* de forma mais fácil e objetiva através de novos elementos que estimulam a inovação e a implementação de novas ideias. O *Canvas* é um desses elementos, que permite criar gráficos e imagens diretamente no navegador através da linguagem *Javascript*. Torna-se necessária uma ferramenta que facilite o trabalho de criação e edição, dado que o procedimento é complexo e trabalhoso.

Softwares de ilustração vetorial como *Flash* e *Illustrator* possuem atualmente o recurso de conversão de seus formatos próprios para o HTML5, porém não são completamente acessíveis, por serem *softwares* pagos e executarem apenas em plataformas específicas. O *Inkscape* é um *software* similar, além de ser um projeto de *software* livre e multiplataforma, permitindo contribuições com melhorias no código e recursos novos. Este trabalho foi proposto como uma extensão para o *Inkscape* que converte seu formato nativo, o SVG, para HTML5. Com isso, pretende prover uma alternativa livre para esse recurso presente nos *softwares* citados, além de permitir o estudo e posterior melhoria do código pela comunidade de desenvolvedores, de forma colaborativa.

4 REFERENCIAL TEÓRICO

Neste capítulo serão apresentados os conceitos necessários à compreensão do funcionamento da ferramenta de conversão *Ink2canvas*. Primeiramente são brevemente introduzidos os padrões HTML5, o elemento *Canvas* e em seguida o SVG, abrangendo suas principais características com exemplos e imagens. Logo após é apresentado o *software* de ilustração *Inkscape*, sua relação com os padrões estudados e por fim, a estrutura de extensões.

4.1 HTML5 CANVAS

4.1.1 HISTÓRICO

O World Wide Web Consortium define *Hipertext Markup Language* (HTML) como a linguagem para descrever a estrutura de páginas Web, que oferece aos autores a possibilidade de publicar na Internet documentos com estruturas bem definidas, incluir recursos multimídia, estabelecer ligações com outros documentos e recuperar informações através de hyperlinks e desenvolver aplicações em conjunto com outras tecnologias (W3C, 2011).

Pilgrim (2010) relata que nos seus primeiros cinco anos (1990 a 1995), a linguagem HTML passou por várias revisões. Em Dezembro de 1997 o W3C lançou oficialmente a versão 4 do HTML, que foi amplamente utilizada e marcou o crescimento da Web nos últimos anos. No ano seguinte o W3C abandonou o desenvolvimento do HTML 4 e começou a trabalhar em um formato equivalente ao XML¹ chamado XHTML².

O W3C reformulou o grupo de trabalho do HTML para criar uma nova suíte de elementos baseados em XML que viria a substituir o HTML 4, já que foi decidido que extendê-lo seria muito difícil. O HTML foi reformulado com base no XML sem adicionar nenhum elemento ou atributo extra, e assim nasceu a especificação do XHTML 1.0. O W3C decidiu continuar investindo no desenvolvimento do XHTML e em formatos baseados em XML, como o XForms³, e

¹<http://www.w3.org/XML/>

²<http://www.w3.org/TR/xhtml1/>

³<http://www.w3.org/MarkUp/Forms/>

já havia iniciado a especificação do XHTML 2.

Segundo o WHATWG (2011), empresas como *Apple*, *Mozilla* e *Opera* tinham a intenção de trabalhar na evolução do HTML ao invés de formatos substitutos, então anunciaram sua intenção de trabalhar sob o nome de um novo grupo chamado *Web Hypertext Application Technology Working Group* (WHATWG), que era baseado em vários princípios centrais como o de que tecnologias devem ser retro-compatíveis. O novo padrão no qual trabalhariam, o HTML5, deveria abranger padrões previamente especificados, como o HTML4, XHTML1 e DOM2⁴. Em 2006 o W3C anunciou o interesse em participar do desenvolvimento do HTML5 e desde então ambos vêm trabalhando na sua especificação, que ainda não foi finalizada.

O HTML5 trás vários novos elementos que facilitam o desenvolvimento de sites interativos e dinâmicos. Pilgrim (2010) cita os recursos de vídeo, áudio, armazenamento de dados local, execução de código *Javascript* em *background*, aplicações *offline*, geolocalização, novos tipos de campos de formulário como email e data, *microdata*, que permite adicionar semântica ao elementos, entre outros. A listagem abaixo é o exemplo mais simples possível de um código HTML5:

```
<!doctype html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Basic Hello World HTML Page</title>
  </head>
  <body>
    Hello World!
  </body>
</html>
```

4.1.2 INTRODUÇÃO AO CANVAS

O padrão HTML5 define um novo tipo de elemento chamado `<canvas>`, que segundo a especificação, provê uma superfície de desenho do tipo *bitmap*, que pode ser usada para desenhar gráficos, animações, jogos e outros recursos visuais instantaneamente, através de comandos da linguagem *Javascript* (WHATWG, 2011). Até então a capacidade de criação de gráficos em HTML estava limitada ao uso de *plugins* como *applets Java* e *Flash*, entre outros modos. O `<canvas>` foi introduzido pela *Apple* para o *Dashboard* do *Mac OS X* e mais tarde implementado no navegador *Safari 1.3*. É suportada no *Firefox* desde a versão 1.5, no *Opera* desde a versão 9 e em todas as versões do *Google Chrome*. Versões do *Internet Explorer* anteriores à 9

⁴<http://www.w3.org/DOM/DOMTR>

não suportam o elemento `<canvas>`, mas pode-se emular seu funcionamento nas versões 6, 7 e 8 com o auxílio do *ExplorerCanvas* ⁵.

Segundo Fulton e Fulton (2011), o objeto *Canvas* opera em modo “imediato”, que refere-se ao modo como ele renderiza os pixels na tela. O *Canvas* redesenha completamente a tela a cada atualização fazendo chamadas aos comandos *JavaScript*. Em outras palavras, mover um objeto de uma posição para outra requer que a tela inteira seja apagada e o objeto seja redesenhado na nova posição. O desenvolvedor deve configurar a tela (o que será exibido) antes que cada frame seja renderizado. Isso diferencia o *Canvas* de tecnologias como o *Flash*, *Silverlight* e *SVG*, que operam em modo “retido”. Nesse modo, o renderizador mantém uma lista de objetos, que são exibidos na tela de acordo com atributos definidos no código (i.e., a posição X, a posição Y, transparência, etc). Isso permite ao desenvolvedor evitar trabalhar com operações de baixo nível, porém reduz o controle sobre a renderização final.

Fulton e Fulton (2011) listam alguns recursos disponíveis no *Canvas*, entre eles os vários tipos de formas, como linhas, curvas, retângulos, polígonos, renderização de texto, exibição e tratamento de imagens, preenchimentos, transformações, transparências, além de ser possível manipular *pixels*. Fulton e Fulton (2011) alertam para o fato do contexto do *Canvas* resumir-se apenas à criação de elementos gráficos. Para criar aplicações completas, é preciso utilizar código *Javascript* para demais funcionalidades, como eventos de mouse e teclado, áudio, cronômetros, funções matemáticas, etc.

De acordo com o WHATWG (2011), em tipos de mídias não-visuais ou navegadores e dispositivos que não suportam o elemento *Canvas*, existe a possibilidade de fornecer um conteúdo alternativo, como meio de prover acessibilidade. Caso contrário o *Canvas* representa um tipo de conteúdo que consiste em uma imagem criada dinamicamente, codificada em Base64⁶. O trecho de código abaixo mostra exemplifica a utilização do *Canvas* com conteúdo alternativo, que é exibido em navegadores sem suporte ao elemento:

```
<canvas id="canvas_id" width="500" height="300">
  Seu navegador não suporta o elemento Canvas
</canvas>
```

A especificação define dois atributos para controlar as dimensões da área de desenho, *width* e *height* (largura e altura, respectivamente, em pixels) (WHATWG, 2011). Quando especificados, devem conter valores inteiros não-negativos válidos. Se algum desses atributos não estiver presente, os valores padrão serão utilizados: 300 para a largura e 150 para a altura. O ele-

⁵<http://code.google.com/p/explorercanvas/>

⁶<http://tools.ietf.org/html/rfc4648>

mento pode ser redimensionado através de CSS⁷, mas durante a renderização a imagem final será distorcida para se adaptar ao novo tamanho.

Fulton e Fulton (2011) explicam que aplicações *Canvas* são diferentes de outras aplicações executadas no navegador. Devido ao fato do *Canvas* exibir seus gráficos em uma região específica da tela, sua funcionalidade é auto-suficiente, ou seja, não interfere com o resto da página, além de permitir outros elementos *Canvas* no mesmo documento.

Flanagan (2010) define a habilidade do *Canvas* gerar gráficos dinamicamente no lado do cliente (i.e.: no navegador) como algo revolucionário, e lista os motivos:

- O código utilizado para produzir gráficos no lado do cliente é tipicamente muito menor que as imagens em si, evitando um maior tráfego de dados.
- Carregar comandos de desenho do servidor para o cliente reduz o processamento do servidor, reduzindo custos de hardware.
- Gerar gráficos no cliente é consistente com a estrutura de aplicações AJAX⁸, na qual o servidor provê os dados e o cliente gerencia a formatação deles.
- O cliente pode rápida e dinamicamente redesenhar gráficos permitindo aplicações que usam intensamente recursos gráficos, como jogos e simulações, que simplesmente seriam impossíveis se cada quadro da animação tivesse de ser carregado a partir de um servidor.

4.1.3 O CONTEXTO DE RENDERIZAÇÃO

O WHATWG (2011) especifica que o elemento `<canvas>` cria uma superfície de desenho que disponibiliza um ou mais contextos de renderização, que são usados para criar e manipular os gráficos. O contexto padrão é o 2D, mas outros contextos podem oferecer outros tipos de renderização. Por exemplo, há um contexto 3D experimental baseado em *OpenGL*, o *WebGL*⁹. Quando o método `canvas.getContext()` do objeto *Canvas* é chamado passando o parâmetro 2D, retorna um objeto *CanvasRenderingContext2d*, que representa uma superfície cartesiana cuja origem (0, 0) fica no canto superior esquerdo da janela, onde o valor x aumenta para a direita e o valor y aumenta para baixo, conforme ilustrado na figura 1. O trecho de código abaixo demonstra esse processo:

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

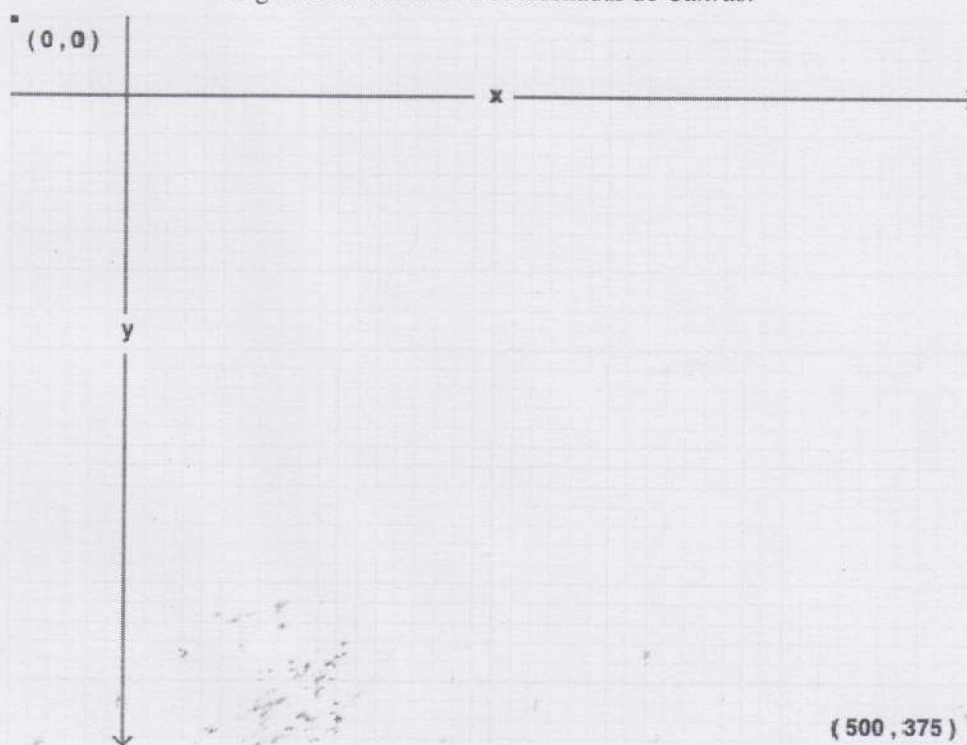
⁷<http://www.w3.org/Style/CSS/>

⁸<http://adaptivepath.com/ideas/ajax-new-approach-web-applications>

⁹https://developer.mozilla.org/en/WebGL/Getting_started_with_WebGL

O objeto *CanvasRenderingContext2d* atribuído à variável *context* no exemplo acima possui os atributos e métodos necessários para criar as formas e efeitos 2D desejados. Segundo Flanagan (2010), cada elemento <canvas> possui apenas um objeto de contexto. Cada nova chamada ao método *canvas.getContext()* retorna o mesmo objeto *CanvasRenderingContext2D*.

Figura 1: O sistema de coordenadas do *Canvas*.



Fonte: (PILGRIM, 2010, p. 60)

Como o *Canvas* opera em modo imediato, Fulton e Fulton (2011) explicam que o estado atual do contexto é armazenado em uma pilha de estados. Cada um desses estados guarda dados sobre o contexto do *Canvas* em um dado momento. O estado é a configuração atual do contexto do *Canvas*, que compreende propriedades de preenchimento e contorno, entre outros. Os comandos de desenho utilizam essas propriedades no momento da criação, até que haja qualquer mudança no contexto.

Fulton e Fulton (2011) listam os tipos de informações que são armazenadas na pilha de estados:

- A matriz de transformação, como por exemplo rotações e translações usando os métodos *context.rotate()* e *context.setTransform()*.
- A região de corte (*clippath*)

- Os valores dos atributos do *Canvas*, que são:
 - *globalAlpha*
 - *globalCompositeOperation*
 - *strokeStyle*
 - *textAlign*, *textBaseline*
 - *lineCap*, *lineJoin*, *lineWidth*, *miterLimit*
 - *fillStyle*
 - *font*
 - *shadowBlur*, *shadowColor*, *shadowOffsetX*, *shadowOffsetY*

4.1.4 FORMAS, CAMINHOS E CURVAS

Fulton e Fulton (2011) descrevem o retângulo como a mais simples forma implementada pelo *Canvas*. Ele pode ser desenhado de três formas:

- **context.fillRect(x, y, width, height)**, que desenha um retângulo preenchido com o valor de preenchimento definido pelo atributo *fillStyle*.
- **context.strokeRect(x,y,width,height)**, que desenha os contornos do retângulo e utiliza os atributos de contorno *strokeStyle*, *lineWidth*, *lineJoin* e *miterLimit*.
- **context.clearRect(x,y,width,height)**, que apaga a área especificada e a torna totalmente transparente.

Cada um desses métodos usa os mesmos parâmetros: *x* e *y* são as posições no sistema de coordenadas e *width* e *height* são a largura e altura, respectivamente. A figura 2 é resultado do seguinte trecho de código, onde três retângulos são desenhados:

```
context.fillStyle = '#000000';
context.strokeStyle = '#ff00ff';
context.lineWidth = 2;
context.fillRect(10,10,40,40);
context.strokeRect(0, 0,60,60);
context.clearRect(20,20,20,20);
```

Segundo Flanagan (2010) esses métodos, por serem simples, desenharam imediatamente no *Canvas*, ao contrário dos caminhos livres, que são compostos de vários métodos e necessitam

Figura 2: Três retângulos foram desenhados: preenchido com a cor preta, um mais externamente com contorno roxo e o último internamente, apagando a área.



Fonte: (FULTON; FULTON, 2011, p. 29)

de comandos extras para finalizar o desenho. Os caminhos, segundo Fulton e Fulton (2011), são um meio para desenhar qualquer forma no *Canvas*. Um caminho é simplesmente uma lista de pontos, e linhas desenhadas entre esses pontos. Um contexto do *Canvas* pode ter apenas um único caminho (que não é armazenado na pilha de estados).

Fulton e Fulton (2011) especificam que os caminhos exigem alguns passos extras: eles precisam ser inicializados com o método *beginPath()* e finalizados com *closePath()*, *stroke()* ou *fill()*. Quando dois pontos se conectam dentro de um caminho, são considerados um subcaminho. Um subcaminho é considerado fechado quando o ponto final se conecta ao ponto inicial. Internamente, caminhos são armazenados como listas de subcaminhos (linhas, retângulos, etc) que juntos formam o desenho desejado. Sempre que o método *beginPath()* é chamado, essa lista é resetada.

Assim que um caminho é inicializado deve-se executar os métodos para de fato desenhar os subcaminhos. Porém, como afirma Pilgrim (2010), nada será visualizado até que algum dos métodos de finalização do caminho seja chamado.

Quando o *Canvas* é inicializado ou o método *beginPath()* é chamado, normalmente usa-se o método *moveTo(x, y)* para configurar o ponto de início de um caminho. Seu uso é similar ao movimento de uma caneta de uma posição qualquer para a posição de início de um desenho, portanto não desenha nada de fato, apenas especifica a posição inicial. Esse método também pode ser utilizado para desenhar caminhos desconectados.

Fulton e Fulton (2011) explicam como funcionam os demais métodos para desenhar caminhos:

- **context.lineTo(x, y)** - Desenha uma linha a partir da posição atual do caminho até o ponto (x, y) especificado em seus parâmetros.
- **context.rect(x, y, width, height)** - Apesar de haver outros métodos para desenhar retângulos diretamente no *Canvas*, há também o método *rect()* que adiciona um retângulo à lista de caminhos. Similar aos métodos estudados previamente, x e y definem a coordenada do

canto superior esquerdo do retângulo, enquanto *width* e *height* definem a largura e altura, respectivamente. Quando este método é executado, o *moveTo(x, y)* é automaticamente chamado com valores (0, 0) para resetar a posição inicial do caminho.

- **context.arc(x, y, radius, startAngle, endAngle, anticlockwise)** - Este método é utilizado para desenhar círculos e arcos. X e Y são as coordenadas do centro do círculo de raio definido pelo parâmetro *radius*. Os parâmetros *startAngle* e *endAngle* definem o ponto inicial e o final do arco, ângulos medidos em radianos. A medida desses ângulos é feita a partir do eixo X. O parâmetro *anticlockwise* é um valor booleano que desenha o arco no sentido anti-horário quando verdadeiro, e vice-versa. Exemplos de uso desse método podem ser vistos na figura 3.
- **context.arcTo(x1, y1, x2, y2, radius)** - Desenha uma linha reta da posição atual do caminho até o ponto dado por (x1, y1), então desenha um arco até o ponto (y1, y2) usando o valor do parâmetro *radius* como raio.

Curvas *Bézier*¹⁰ são mais flexíveis do que arcos, definidas no espaço 2D por um ponto inicial, um ponto final e um ou mais pontos de controle, que determinam a curvatura. Uma curva *Bézier* normal usa dois pontos de controle, enquanto a quadrática usa apenas um. A figura 3 ilustra duas curvas e seus pontos de controle.

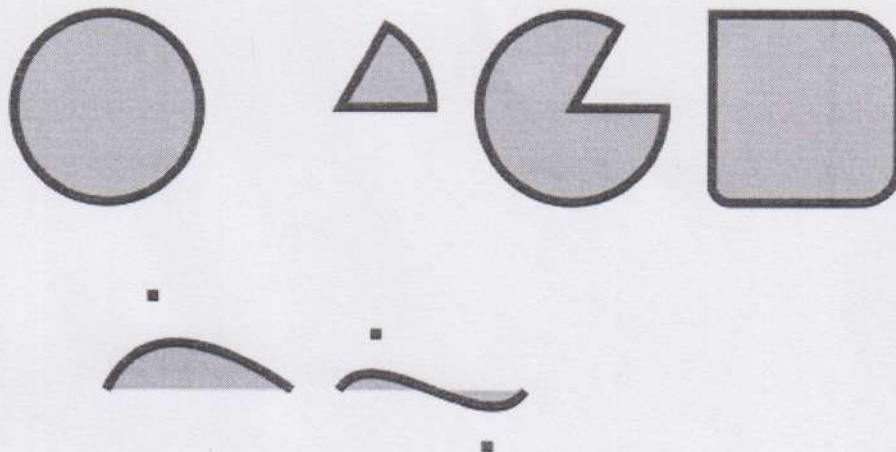
- **context.quadraticCurveTo(cpx, cpy, x, y)** - É o método mais simples, que cria uma curva a partir do ponto atual do caminho até o ponto (x, y), tendo como ponto de controle os parâmetros *cpx* e *cpy*.
- **context.bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)** - Cria uma curva a partir do ponto atual do caminho até o ponto (x, y), tendo como pontos de controle os parâmetros (cp1x, cp1y) e (cp2x, cp2y). Este tipo de curva oferece mais opções de formas por ter dois pontos de controle.

Após traçar o caminho, opcionalmente usa-se o método *closePath()* para de fato finalizar o desenho e exibí-lo. Este método tenta fechar automaticamente o caminho desenhando uma linha reta da posição atual para a inicial, se o caminho não tiver sido fechado explicitamente.

Os métodos *fill()* e *stroke()* são importantes pois são os responsáveis por de fato desenhar o caminho, preenchendo e contornando os objetos e são chamados após a finalizado do caminho. O método *fill()* realiza duas coisas importantes nesse momento: preenche o caminho com o

¹⁰<http://mathworld.wolfram.com/BezierCurve.html>

Figura 3: Exemplos de formas desenhadas com comandos de caminho. Note os pontos de controle nas curvas Bézier na parte inferior.



Fonte: (FLANAGAN, 2010, p. 21)

valor do atributo *fillStyle* e fecha automaticamente o caminho com uma linha reta da mesma forma que o método *closePath()*, exceto pelo fato de que ele não traça o contorno dessa linha. O método *stroke()* é similar, mas usa o valor do atributo *strokeStyle* e não fecha caminhos abertos.

4.1.5 ATRIBUTOS GRÁFICOS

Flanagan (2010) lista 15 atributos gráficos que configuram o estado do *Canvas*, ou seja, são aplicados ao contexto de renderização. Todos eles fazem parte do objeto *CanvasRenderingContext2D* explicado anteriormente. A partir do momento em que um atributo recebe um valor, todos os objetos a serem desenhados em seguida serão formatados com o novo valor, a menos que outro valor seja definido novamente.

Fulton e Fulton (2011) destacam que atributos relacionados a cores utilizam o padrão de cores do CSS¹¹, como os hexadecimais ("#000000"), literais ("black") e RGB (rgb(0, 0, 0)).

- **context.fillStyle** - Define a cor ou estilo que serão utilizados para o preenchimento de um objeto, utilizado pelo método *fill()*. Seu valor padrão é "#000000". Exemplo:

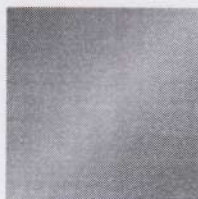
```
context.fillStyle = "#FF0000";
```

- **context.strokeStyle** - Define a cor ou estilo que serão utilizados para o contorno de um objeto, utilizado pelo método *stroke()*. Seu valor padrão é "#000000".

¹¹<http://www.w3.org/Style/CSS/>

O preenchimento e o contorno podem ter cores sólidas ou gradientes. Fulton e Fulton (2011) citam os dois tipos de gradientes que o *Canvas* suporta: linear e radial. Pode-se controlar as cores e a direção do gradiente adicionando pontos de controle ao longo do objeto que será preenchido, como pode ser visto na figura 4, onde foi aplicado um gradiente com três cores na direção diagonal.

Figura 4: Exemplo de gradiente linear.

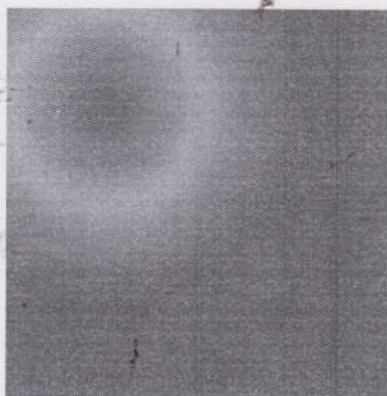


Fonte: (FULTON; FULTON, 2011, p. 58)

Os gradientes lineares podem ser horizontais, verticais ou diagonais. Para criar um gradiente é preciso declarar uma variável que vai referenciar o gradiente, através do método *createLinearGradient(x1, y1, x2, y2)*. Os parâmetros são o ponto inicial e o ponto final do gradiente. O trecho de código abaixo cria um objeto do tipo gradiente linear:

```
var gradiente = context.createLinearGradient(0, 0, 100, 0);
```

Figura 5: Exemplo de gradiente radial com mais de duas cores.



Fonte: (FULTON; FULTON, 2011, p. 59)

O gradiente radial, como pode ser visto na figura 5, é criado de forma semelhante, mas com parâmetros adicionais: *createRadialGradient(x1,y1,r1,x2,y2,r2)*. Os três primeiros parâmetros definem um círculo no ponto (x1, y1) com raio r1 e os demais definem um círculo no ponto (x2, y2) com raio r2:

```
var gradiente = context.createRadialGradient(50,50,25,50,50,100);
```

Uma vez que o gradiente é criado, adicionam-se cores através do método *addColorStop(position, color)*. O primeiro argumento indica a posição relativa ao início do gradiente e deve assumir um valor entre 0.0 e 1.0, enquanto o segundo argumento recebe uma propriedade de cor, de acordo com o padrão de cores CSS:

```
gradiente.addColorStop(0, '#FF0000');
gradiente.addColorStop(.5, '#00FF00');
gradiente.addColorStop(1, '#FF0000');
```

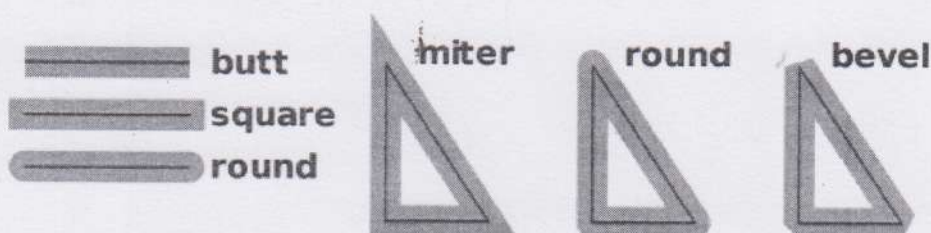
Após a criação do objeto gradiente e da definição de suas cores e posições, o atributo recebe a referência a esse objeto como propriedade de preenchimento ou contorno. Esse mesmo objeto pode ser aplicado várias vezes.

```
context.fillStyle = gradiente;
```

Fulton e Fulton (2011) listam os demais atributos de estilo:

- **context.lineWidth** - Define a espessura da linha, cujo valor padrão é igual a 1,0.
- **context.lineCap** - É o estilo da ponta de uma linha. A figura 6 ilustra os tipos que pode assumir, que são:
 - *butt* - O valor padrão; uma superfície perpendicular ao limite da linha.
 - *round* - Um semicírculo com o diâmetro igual à largura da linha.
 - *square* - Um retângulo com largura igual à metade da largura da linha.

Figura 6: Formas com diferentes atributos de linha aplicados.

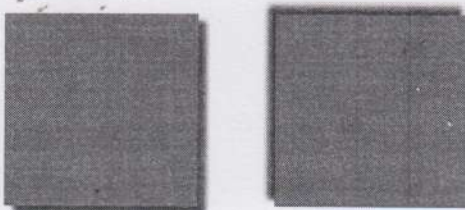


Fonte: (FLANAGAN, 2010, p. 29)

- **context.lineJoin** - É a junção entre duas linhas, exemplificado na figura 6, onde é criado um triângulo preenchido, que pode assumir os seguintes valores:
 - *miter* - O valor padrão; o atributo *miterLimit* define o espaçamento máximo entre as linhas, cujo valor padrão é igual a 10.
 - *bevel* - Cria uma junção diagonal.
 - *round* - Cria um junção arredondada.
- **context.font** - Define as propriedades de fonte, utilizando um padrão similar ao especificado na CSS. Exemplo:


```
context.font = '20px Times New Roman';
```
- **context.textAlign** - Define o alinhamento horizontal do texto.
- **context.textBaseline** - Define o alinhamento vertical do texto.
- **context.globalAlpha** - Especifica a transparência que será atribuída a todos os *pixels*.
- **context.shadowBlur** - Define o quão desfocada a sombra será. Seu valor padrão é 0.
- **context.shadowColor** - A cor da sombra, seguindo o padrão de cores CSS.
- **context.shadowOffsetX** e **context.shadowOffsetY** - Define o deslocamento horizontal e vertical da sombra em relação ao objeto, conforme ilustrado na figura 7.

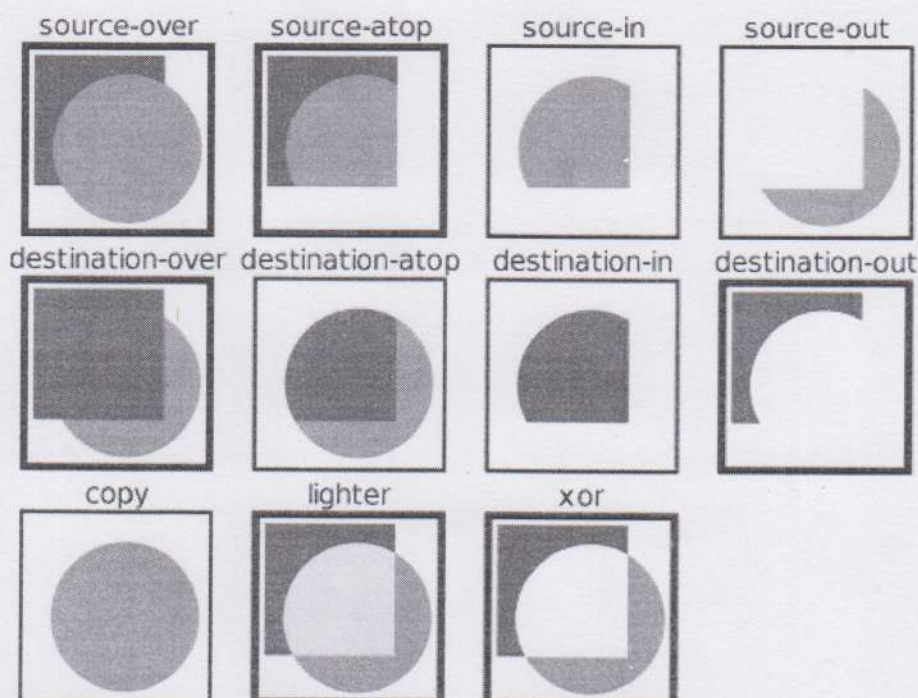
Figura 7: Exemplo de sombras aplicadas a quadrados, com variações das propriedades *shadowOffsetX* e *shadowOffsetY*.



Fonte: Adaptado de (FULTON; FULTON, 2011, p. 66)

- **context.globalCompositeOperation** - Permite modificar e compor desenhos a partir de objetos que estejam se sobrepondo, além de definir a ordem em que os objetos são desenhados. A figura 8 ilustra os vários tipos de composição que podem ser aplicados.

Figura 8: Os vários tipos de atributos de composição aplicados a um quadrado e um círculo que se sobrepõem.



Fonte: (FLANAGAN, 2010, p. 41)

4.1.6 TRANSFORMAÇÕES

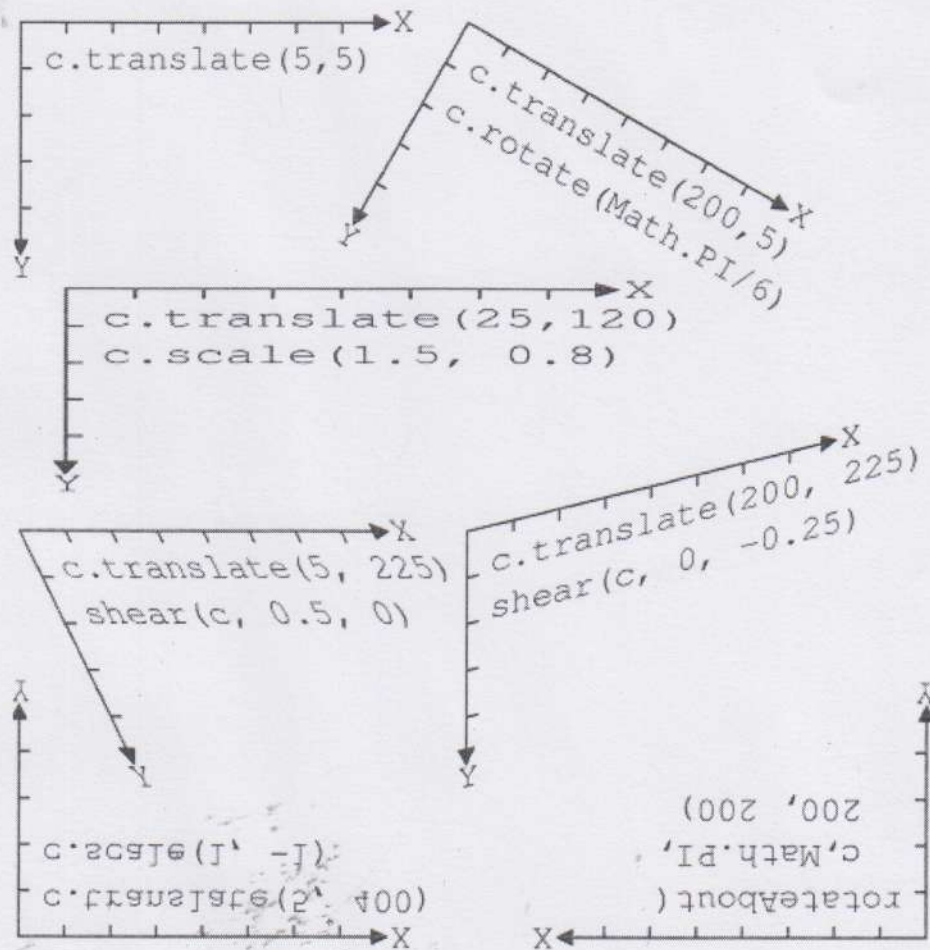
Fulton e Fulton (2011) definem transformações como ajustes matemáticos de propriedades físicas das formas desenhadas no *Canvas*. Essas transformações são feitas através de matrizes matemáticas de operações.

De acordo com Flanagan (2010), todo *Canvas* tem uma matriz de transformação como parte de seu estado. Essa matriz define o atual sistema de coordenadas do *Canvas*. Na maioria das operações, quando são especificadas as coordenadas de um ponto, este ponto é definido no sistema de coordenadas atual, e não no sistema de coordenadas padrão. A matriz de transformação atual é usada para converter as coordenadas especificadas para as coordenadas equivalentes no sistema padrão. Na figura 9 são exemplificados vários tipos de transformação.

Flanagan (2010) explica que o método *setTransform()* é usado para definir a matriz de transformação diretamente, mas operações de transformação são mais fáceis se definidas numa sequência de comandos específicos, como translações, rotações, entre outros.

Os seguintes métodos podem ser utilizados:

Figura 9: Exemplos de transformações e os comandos que permitem aplicar distorções no *Canvas*.



Fonte: (FLANAGAN, 2010, p. 15)

- **context.translate(x, y)** - Move a origem do sistema de coordenada para o ponto (x, y). Um objeto desenhado no ponto (0, 0) será desenhado na verdade no ponto (x, y).
- **context.rotate(x)** - Rotaciona o *Canvas* x graus radianos no sentido horário.
- **context.scale(x, y)** - Distorce o *Canvas* horizontal e verticalmente.
- **context.transform(m11, m12, m21, m22, dx, dy)** - Este método aplica a matriz de transformação especificada por seus parâmetros diretamente na matriz atual do *Canvas*.
- **context.setTransform(m11, m12, m21, m22, dx, dy)** - Equivalente ao método transform, mas primeiro reseta a matriz de transformação atual para os valores padrão e executa o método transform passando os parâmetros especificados.

4.1.7 SALVANDO E RESTAURANDO O CONTEXTO

De acordo com Fulton e Fulton (2011), o contexto do *Canvas* possui um estado, que é uma representação de todos os estilos e transformações que foram aplicados ao *Canvas*. Os estados do *Canvas* são armazenados em uma pilha. O *Canvas* possui dois métodos para manipular seu estado: *save()* e *restore()*.

- **context.save()** Cada vez que esse método é chamado, o estado atual do *Canvas* é inserido na pilha. É utilizado para guardar temporariamente um estado do *Canvas* de forma que desenhos com diferentes propriedades possam ser renderizados, sem alterar o estado comum do *Canvas*. Esse método pode ser chamado várias vezes.
- **context.restore()** Este método é utilizado para retirar o último estado salvo da pilha, retornando o *Canvas* ao estado anterior.

4.1.8 ÁREA DE CLIP

Flanagan (2010) define a área de *clip* como uma região onde nada será desenhado fora de sua área, agindo como uma máscara. Por padrão o *Canvas* possui uma área de *clip* que compreende toda a sua área, ou seja, não ocorre o mascaramento.

Figura 10: O *clip* restringe a exibição dos objetos à área do círculo.



Fonte:

https://developer.mozilla.org/samples/canvas-tutorial/6_2_canvas_clipping.html.
Acesso em: 27 Nov. 2011

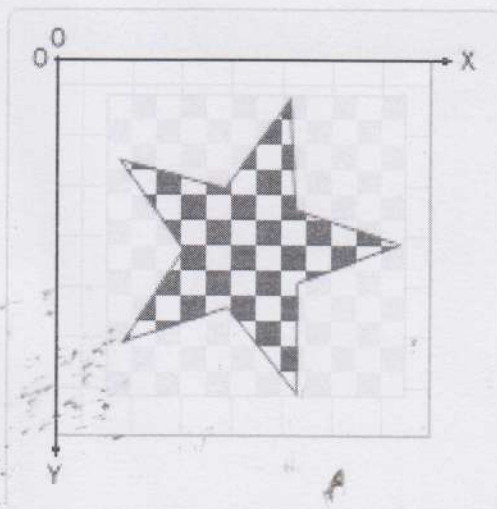
O método *clip()* é utilizado para criar uma nova área de *clip*. Após ser chamado, esse método altera o estado do *Canvas*, e a partir de então todos os desenhos aparecerão limitados a

essa área, especificada por um método de desenho qualquer. No exemplo abaixo é criada uma área de *clip* com um círculo, ilustrada na figura 10, onde tudo que estiver fora da área circular é oculto.

```
context.beginPath();
context.arc(0,0,60,0,Math.PI*2,true);
context.clip();
```

Utilizam-se os métodos *context.save()* e *context.restore()* em casos em que se deseja aplicar a área de *clip* somente a alguns objetos do *Canvas*. A figura 11 ilustra melhor o conceito de *clip*.

Figura 11: O *clip* oculta tudo que não estiver em sua área.



Fonte: <<http://learnhtml5.info/2011/10/clipping-paths/>>. Acesso em: 27 Nov. 2011

4.1.9 TEXTO

Para Fulton e Fulton (2011), a API de texto do *Canvas* permite que desenvolvedores renderizem texto no HTML de maneiras que seriam impossíveis antes de sua invenção. Apesar de ser a parte mais simples do *Canvas*, é um dos últimos recursos a serem implementados nos navegadores.

A forma mais simples de renderizar texto no *Canvas* é definir o valor do atributo *context.font*, utilizando o padrão de valores de fontes CSS¹²: *font-style*, *font-weight*, *font-size* e *font-face*. O formato básico que o atributo *context.font* pode assumir é dado abaixo:

¹²<http://www.w3.org/Style/CSS/>

[font style] [font weight] [font size] [font face]

Os exemplos abaixo demonstram como utilizar o atributo *context.font*:

```
context.font = "50px serif";
context.font = "italic bold 24px serif";
context.font = "normal lighter 50px cursive";
```

A cor do texto é definida pelo atributo estudado anteriormente *context.fillStyle*. Após definir as propriedades de estilo, utiliza-se o método *context.fillText()*, usando como parâmetros o texto a ser escrito e as posições x e y, conforme exemplo abaixo:

```
context.font = "50px serif"
context.fillStyle = "#FF0000";
context.fillText ("Exemplo de texto", 100, 80);
```

Se uma fonte não for especificada, será usada a padrão, *sans-serif* com 10 pixels.

Similarmente ao método *context.fillText()*, existe o *context.strokeText()*. A única diferença é que este último renderiza o texto apenas com contornos, e utiliza a propriedade *context.strokeStyle*.

Figura 12: Atributos de texto *textAlign* e *textBaseline*.

	start	left	center	right	end
top	Abcefg	Abcefg	Abcefg	Abcefg	Abcefg
hanging	Abcefg	Abcefg	Abcefg	Abcefg	Abcefg
middle	Abcefg	Abcefg	Abcefg	Abcefg	Abcefg
alphabetic	Abcefg	Abcefg	Abcefg	Abcefg	Abcefg
ideographic	Abcefg	Abcefg	Abcefg	Abcefg	Abcefg
bottom	Abcefg	Abcefg	Abcefg	Abcefg	Abcefg

Fonte: (FLANAGAN, 2010, p. 31)

O atributo *context.textAlign* representa o alinhamento horizontal do texto baseado em sua posição x. A figura 12 ilustra as diferentes formas que o texto apresenta de acordo com os valores permitidos, que são:

- **center** - Alinhamento ao centro do texto

- **start** - O texto é exibido depois da posição y.
- **end** - Todo o texto é exibido antes da posição y.
- **left** - Alinhamento à esquerda do texto.
- **right** - Alinhamento à direita do texto.

4.1.10 IMAGENS

Segundo Fulton e Fulton (2011), a API de renderização de imagens é tão robusta quanto a de desenho. Com essa API é possível carregar uma imagem e aplicá-la diretamente ao *Canvas*, além de ser possível editá-la através dos recursos já estudados, como as transformações e *clips*. A API do *Canvas* permite acessar uma imagem através do objeto *DOM Image*¹³ utilizando o método *drawImage()*. A imagem pode ser definida no HTML da seguinte forma:

```

```

A imagem pode ser criada através de *Javascript*, utilizando a classe *Image*:

```
var imagem = new Image();
```

Dessa forma é preciso definir um valor para o atributo *src* do objeto *Image*:

```
imagem.src = "imagem.png";
```

Assim que uma imagem é carregada, pode-se aplicá-la no *Canvas* de várias maneiras. O método *drawImage()* é utilizado para isto, e por ser sobrecarregado, possui três tipos de conjuntos de parâmetros, permitindo diferentes formas de manipular a disposição dos *pixels* no *Canvas*.

- **drawImage(Image, dx, dy)** - Este método recebe três parâmetros: um objeto *Image* e a posição x e y do canto superior-esquerdo da imagem. O código abaixo é um exemplo de como este método seria aplicado:

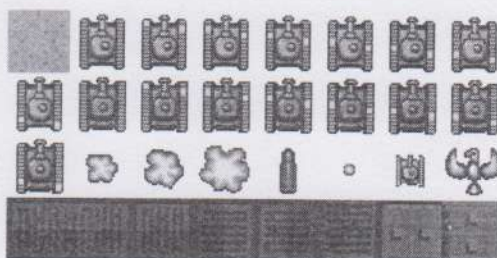
```
context.drawImage(imagem, 0, 0);
```

- **drawImage(Image, dx, dy, dw, dh)** - Permite modificar as dimensões da imagem ao renderizá-la, através dos parâmetros dw (largura) e dh (altura).

¹³<http://www.w3.org/TR/DOM-Level-2-HTML/html.html#ID-17701901>

- **drawImage(Image, sx, sy, sw, sh, dx, dy, dw, dh)** - Os parâmetros extras desse método permitem recortar uma parte da imagem e renderizar apenas essa parte especificada. Fulton e Fulton (2011) citam os *sprites* de jogos como um exemplo de utilização desse método. Jogos geralmente utilizam uma única imagem com todas as posições e desenhos de um personagem, como visto na figura 13. Esse método é utilizado para mapear as posições de cada personagem e posição, permitindo que seja utilizada apenas uma imagem para vários personagens, eliminando requisições desnecessárias a vários arquivos de imagens.

Figura 13: Imagem com vários *sprites* utilizados em um jogo.



Fonte: (FULTON; FULTON, 2011, p. 132)

Flanagan (2010) afirma que é possível extrair o conteúdo do *Canvas* como uma imagem usando o método `canvas.toDataURL()`. O detalhe importante sobre este método é que pertence ao objeto *Canvas*, ao contrário dos outros, que pertencem ao objeto *CanvasRenderingContext2D*. O método `canvas.toDataURL()` é chamado sem parâmetros e retorna o conteúdo do *Canvas* como uma imagem PNG, codificada em uma *string*, que pode ser utilizada em um elemento `<img/>` do HTML. O exemplo abaixo ilustra esse procedimento.

```
var img = document.createElement("img");
img.src = canvas.toDataURL();
document.body.appendChild(img);
```

4.2 SVG

4.2.1 DEFINIÇÃO

De acordo com Eisenberg (2002), *Scalable Vector Graphics*, ou SVG, é um padrão de documento baseado em XML e definido pelo W3C, que permite representar informações gráficas 2D de forma compacta e portátil. Como SVG é uma aplicação XML, os dados são armazenados em texto puro, e traz as vantagens do XML, como a interoperabilidade, clareza e portabilidade.

Há atualmente um crescente interesse no SVG, e várias ferramentas para gerar e visualizar arquivos SVG estão disponíveis. Segundo a especificação o SVG permite três tipos de objetos gráficos: formas vetoriais (i.e linhas e curvas), imagens e texto (W3C, 2011). O código abaixo é um exemplo de documento SVG básico que desenha um retângulo:

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.0//EN"
  "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
<svg width="140" height="170">
  <rect x="10" y="20" width="400" height="200"/>
</svg>
```

4.2.2 HISTÓRICO

A especificação do SVG relata que durante os anos 80 a página com a descrição da linguagem *PostScript*¹⁴ desenvolvida pela *Adobe Systems Inc.* criou um modo de descrever imagens de uma maneira que poderia ser redimensionada para se adaptar à resolução do dispositivo, geralmente uma impressora (W3C, 2011). Era natural que uma solução vetorial baseada na web fosse desenvolvida. Em 1998 a *Microsoft* desenvolveu uma linguagem baseada em XML, chamada VML¹⁵ (*Vector Markup Language*). A VML contém a maioria dos recursos que o SVG, apesar de poucos desenvolvedores a terem adotado e a *Microsoft* ter abandonado o projeto (W3C, 2011).

De acordo com Eisenberg (2002), em 1998 o W3C formou um grupo de trabalho para desenvolver uma representação de gráficos vetoriais na forma de uma aplicação XML. Segundo a especificação, o primeiro rascunho público do SVG foi lançado pelo W3C em Fevereiro de 1999 (W3C, 2011). Durante os anos que se seguiram o interesse no uso de gráficos vetoriais cresceu bastante e várias ferramentas foram desenvolvidas por empresas como *IBM* e *Corel*.

4.2.3 SISTEMAS GRÁFICOS

Eisenberg (2002) afirma que os dois maiores tipos de representação de gráficos em computadores são o *bitmap* (*raster*) e vetores, comparados na figura 14.

Em imagens rasterizadas, uma imagem é representada como uma matriz retangular de pixels, onde cada um é representado por sua cor RGB (*Red, Green, Blue*) ou um índice numa lista de cores. Essa série de *pixels* é chamada de *bitmap* (mapa de bits), geralmente armazenada

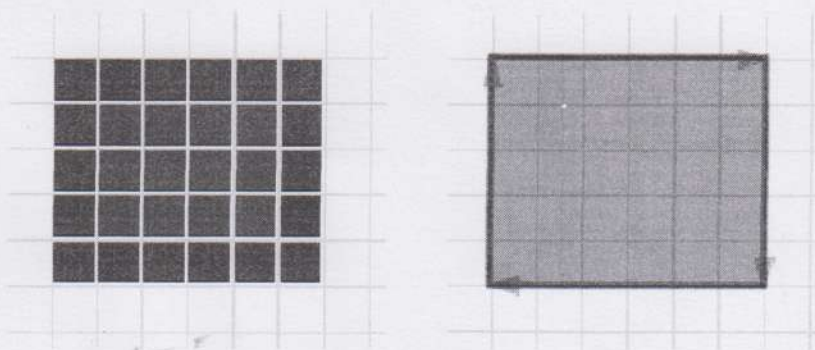
¹⁴<http://www.adobe.com/products/postscript/>

¹⁵<http://www.w3.org/TR/NOTE-VML>

em um formato compactado. Os programas de visualização de imagens simplesmente descompactam as informações e as exibem. O *Canvas* é um exemplo de representação de imagem rasterizada, pois gera um bitmap no formato PNG (FULTON; FULTON, 2011).

O segundo tipo de representação gráfica citado por Eisenberg (2002) é o vetorial, onde a imagem é representada como uma série de formas geométricas. Ao invés de receber um conjunto completo de *pixels*, um programa de visualização vetorial recebe comandos para desenhar formas geométricas em pontos específicos do sistema de coordenadas. O SVG é um formato de representação vetorial.

Figura 14: Uma imagem *bitmap* é composta por uma matriz de *pixels* (esquerda), enquanto uma imagem vetorial é composta de valores utilizados por comandos de desenho (direita).



Fonte: Adaptado de (EISENBERG, 2002, p. 13,14)

4.2.4 FORMAS BÁSICAS

Eisenberg (2002) lista as formas básicas que compõem o SVG:

- **Linhas** - O SVG permite desenhar linhas com o elemento `<line>`. É necessário especificar os pontos iniciais e finais, através dos atributos `x1`, `y1`, `x2` e `y2`, conforme exemplo abaixo, que desenha uma linha horizontal:

```
<line x1="40" y1="20" x2="80" y2="20" />
```

- **Retângulos** - Os retângulos são as formas básicas mais simples do SVG. São especificados as coordenadas `x` e `y` do canto superior esquerdo do retângulo, largura e altura. O código abaixo exemplifica um retângulo com largura de 30 e altura de 50 *pixels* na posição (10, 10).

```
<rect x="10" y="10" width="30" height="50" />
```

O SVG permite criar retângulos com bordas arredondadas, utilizando os atributos *rx* e *ry*:

```
<rect x="40" y="10" width="20" height="40" rx="10" ry="10"/>
```

- **Círculos e elipses** - Para desenhar círculos e elipses, deve-se usar o elemento `<circle>` e o `<ellipse>`. O círculo possui os atributos *cx*, *cy* e *r*, para especificar as coordenadas do centro do círculo e o raio, respectivamente. A elipse utiliza dois atributos para definir o raio horizontal e o vertical, *rx* e *ry*.

Exemplo de círculo com raio 20 e com centro no ponto (80, 30):

```
<circle cx="80" cy="30" r="20" />
```

Exemplo de elipse com raio horizontal 20, raio vertical 10 e com centro no ponto (80, 80):

```
<ellipse cx="80" cy="80" rx="20" ry="10" />
```

- **Polígonos** - O elemento `<polygon>` permite descrever uma série de pontos que definem uma forma geométrica. O exemplo abaixo cria um paralelogramo com os pontos (15, 10), (55, 10), (45, 20) e (5, 20).

```
<polygon points="15,10 55,10 45,20 5,20" />
```

O elemento `<polyline>` é similar ao `<polygon>`, mas ao contrário dele, seu caminho não é automaticamente fechado, então é útil para desenhar séries de linhas retas conectadas.

Exemplo:

```
<polyline points="5 20, 20 20, 25 10, 35 30, 45 10" />
```

4.2.5 CAMINHOS LIVRES

Segundo Eisenberg (2002) todas as formas básicas descritas acima são versões mais específicas do elemento `<path>`, que tem uma aplicação geral, e recomenda utilizar os elementos citados anteriormente por ajudarem a deixar o documento SVG mais legível e estruturado. O elemento `<path>` desenha qualquer tipo de forma através de uma série de linhas, arcos e curvas conectadas. Todos os dados que descrevem o desenho são armazenados no atributo *d*. Os dados consistem em comandos definidos por letras como *M* (*moveTo*) ou *L* (*lineTo*), seguidos de suas respectivas coordenadas. São eles:

- **moveto** - Todo caminho deve começar com o comando *moveto*. A letra do comando é um *M* seguido das coordenadas *X* e *Y*, separados por espaços. Este comando marca a posição inicial da “caneta” que cria o desenho.
- **lineto** - O comando *moveto* pode ser seguido de um ou mais comandos *lineto*, denotado pela letra *L* e coordenadas *X* e *Y*.
- **closepath** - Este comando fecha o desenho automaticamente desenhando uma linha reta do ponto inicial ao ponto final, caso seja necessário. É denotado pela letra *Z*.

O exemplo abaixo exemplifica os três comandos introduzidos até agora:

```
<path d="M 40 60, L 10 60, L 40 42.68, M 60 60, L 90 60, L 60 42.68 Z" />
```

- **arc** - O arco é a secção de uma elipse, ou uma curva, que conecta dois pontos e seu comando é dado pela letra *A*. Apesar de ser visualmente simples, exige mais parâmetros que os demais comandos, como pode ser visto no exemplo abaixo:

```
<path d="M 125,75 A 100,50 0 0,0 225,125" />
```

- **bezier** - As curvas Bézier são formas mais convenientes de desenhar curvas, como explicado anteriormente. A forma mais simples é a curva quadrática, representada pela letra *Q*. No exemplo abaixo é desenhado uma curva Bézier do ponto (30, 75) ao (300, 120) com o ponto de controle em (240, 30):

```
<path d="M 30 75 Q 240 30, 300 120"/>
```

O SVG permite definir curvas Bézier cúbicas, que possuem um ou dois pontos de controle, e são definidas pela letra *C*. No exemplo abaixo descreve-se uma curva do ponto (20, 80) ao (200, 120) com pontos de controle (50, 20) e (150, 60):

```
<path d="M 20 80 C 50 20, 150 60, 200 120"/>
```

4.2.6 TEXTO

A especificação do SVG define que textos devem ser renderizados utilizando o elemento `<text>`, da mesma forma que os outros elementos gráficos (W3C, 2011). Transformações, preenchimento, clips e outros recursos são aplicados da mesma forma. O simples elemento `<text>`

renderiza apenas uma linha. Caso seja necessário renderizar mais de uma linha, utilizam-se outros elementos `<text>` em posições diferentes, quebras de linha manuais ou vários elementos-filho `<tspan>`.

Eisenberg (2002) explica que a forma mais simples de utilização do elemento `<text>` requer apenas dois atributos X e Y, para definir a posição da base do texto. Exemplo:

```
<text x="20" y="30">Texto mais simples</text>
```

Através do elemento `<tspan>` é possível aplicar estilos, transformações e posições diferentes para caracteres, palavras ou linhas distintas, além de permitir textos multilinha. O posicionamento é realizado com os atributos *dx* e *dy*, que permitem adicionar formatações como subscrito e sobrescrito. No trecho de código abaixo são aplicadas propriedades de fonte negrito e itálico a um elemento de texto:

```
<text x="10" y="30" style="font-size:12pt;">
  Texto em
  <tspan style="font-style:italic">itálico</tspan>,
  normal, e
  <tspan style="font-weight:bold" dy="20">negrito e subscrito</tspan>.
</text>
```

4.2.7 ESTILO

Segundo a especificação o SVG utiliza propriedades de estilo para descrever as várias características visuais dos elementos e como eles serão renderizados (W3C, 2011). Entre estas características estão a cor de preenchimento, contorno, gradientes, fonte, tamanho do texto e áreas de clip. O SVG, assim como o *Canvas*, segue o mesmo padrão de estilo CSS já citado, exceto por algumas regras específicas do SVG.

O SVG permite formatar seus objetos das mesmas maneiras que o HTML5, como por exemplo, com folhas de estilo externas (CSS), e o elemento `<style>`. Porém, o atributo *style* é mais utilizado em softwares de renderização, pois permite aplicar propriedades específicas para cada elemento. A declaração das propriedades no atributo *style* segue a regra ‘‘nome: valor;’’ conforme o padrão CSS, separados por vírgula. No exemplo abaixo um elemento foi definido com preenchimento vermelho e contorno preto:

```
<rect
x="10" y="20"
width="400" height="200"
style="fill:#FF0000; stroke: #000000"
/>
```

4.2.8 TRANSFORMAÇÕES

De forma análoga às transformações aplicadas no sistema de coordenadas do Canvas, o SVG permite aplicar transformações no espaço 2D através do atributo *transform*. Os tipos de transformações podem ser aplicados através do uso de funções de transformação. A especificação define seis tipos: *matrix*, *translate*, *scale*, *rotate*, *skewX* e *skewY*¹⁶ (W3C, 2011). O exemplo abaixo mostra uma translação sendo aplicada a um retângulo:

```
<rect x="10" y="10"
width="110" height="240"
transform="translate(30, 50)"
/>
```

4.2.9 GRUPOS

Grupos são definidos pelo elemento `<g>` e servem para agrupar um ou mais elementos, inclusive outros grupos. Além da função natural de agrupar elementos permitindo que estilos e transformações sejam aplicados a todos os elementos internos, podem ser utilizados como camadas de visualização ou páginas em *softwares* de visualização (EISENBERG, 2002).

Exemplo de uso do elemento `<g>`:

```
<g id="casa" style="fill: white; stroke: black;">
  <rect x="6" y="50" width="60" height="60"/>
  <polyline points="6 50, 36 9, 66 50"/>
  <polyline points="36 110, 36 80, 50 80, 50 110"/>
</g>
```

Transformações e estilos podem ser aplicados a um grupo, permitindo que todos os elementos internos sofram a mesma alteração. Cada elemento desse grupo pode ter sua própria transformação e estilo, assim caracterizando um efeito cumulativo.

4.2.10 REFERÊNCIAS E DEFINIÇÕES

Eisenberg (2002) afirma que gráficos complexos podem ter vários elementos repetidos. O SVG possui o elemento `<use>` para referenciar objetos já criados, evitando repetição de código e evitando arquivos maiores. Esse elemento utiliza o atributo *href* em conjunto com o padrão *xlink*¹⁷ para referenciar elementos do documento.

¹⁶<http://www.w3.org/TR/SVG/coords.html#TransformAttribute>

¹⁷<http://www.w3.org/TR/xlink/>

No exemplo abaixo o elemento faz uma referência ao elemento com id “casa” e o renderiza na posição (70, 100):

```
<use xlink:href="#casa" x="70" y="100"/>
```

A desvantagem de utilizar apenas o <use> é que o objeto referenciado precisa estar presente no documento, sua posição precisa ser conhecida a princípio para ser usada como base e as cores e estilos não podem ser sobrescritas. O elemento <defs> evita esses problemas, armazenando a definição de um objeto sem a necessidade de criá-lo sem obrigatoriamente exibi-lo. A especificação recomenda que qualquer objeto possa ser reutilizado deva ser definido no elemento <defs> (W3C, 2011). No exemplo abaixo, define-se um grupo e são feitas duas referências:

```
<svg width="240px" height="240px">
  <!-- Definindo o grupo -->
  <defs>
    <g id="casa" style="stroke: black;">
      <rect x="0" y="41" width="60" height="60" />
      <polyline points="0 41, 30 0, 60 41" />
      <polyline points="30 101, 30 71, 44 71, 44 101" />
    </g>
  </defs>

  <!-- Usando o grupo -->
  <use xlink:href="#casa" x="0" y="0" style="fill: #cfc;" />
</svg>
```

O elemento <defs> é utilizado para guardar definições de gradientes, áreas de *clips* e vários outros recursos que podem ser reutilizados por vários elementos.

4.2.11 CLIPS

Eisenberg (2002) explica que em certos casos é necessário visualizar apenas parte de um desenho. Por exemplo, para criar um desenho que simule uma visão de um binóculo e exibir apenas o que estiver na área dos óculos. O SVG oferece o recurso de *clip* para este tipo de situação. A área de desenho do SVG é por si só uma área recortada, ou seja, tudo que estiver fora dela não será visualizado. É possível definir áreas de *clip* e reutilizá-las por todo o documento, por meio do elemento <clipPath> nas definições do SVG <defs>, conforme exemplificado abaixo:

```

<!-- Define uma área de clip retangular -->
<defs>
  <clipPath id="rectClip">
    <rect id="rect1" x="15" y="15" width="40" height="45" />
  </clipPath>
</defs>

<!-- Apenas uma parte deste retangulo será exibida -->
<rect x="10" y="10" width="200" height="400" clip-path="url(#rectClip)"/>

```

O atributo *clip-path* (que também pode vir como uma propriedade do atributo *style*) guarda uma referência ao objeto que define a área de *clip*. Qualquer elemento pode reutilizar essa área, já que foi definida no elemento `<defs>`.

4.3 O INKSCAPE E A ESTRUTURA DE EXTENSÕES

Jurković e Scala (2011) afirmam que o *Inkscape*¹⁸ é frequentemente mencionado como um dos melhores exemplos de *softwares* livres disponíveis hoje, considerado uma alternativa a *softwares* proprietários como *Adobe Illustrator* e *Corel Draw*. À medida que gráficos vetoriais se tornam cada vez mais importantes e o W3C inicia sua transição para tecnologias HTML5, os principais navegadores aperfeiçoam seu suporte ao padrão SVG, dada a atratividade dos gráficos escaláveis, de alta definição e tridimensionais, além da possibilidade de criação de sites interativos e jogos sem utilizar tecnologias como o *Flash*. A interface do *Inkscape* é bastante simples, como mostra a figura 15.

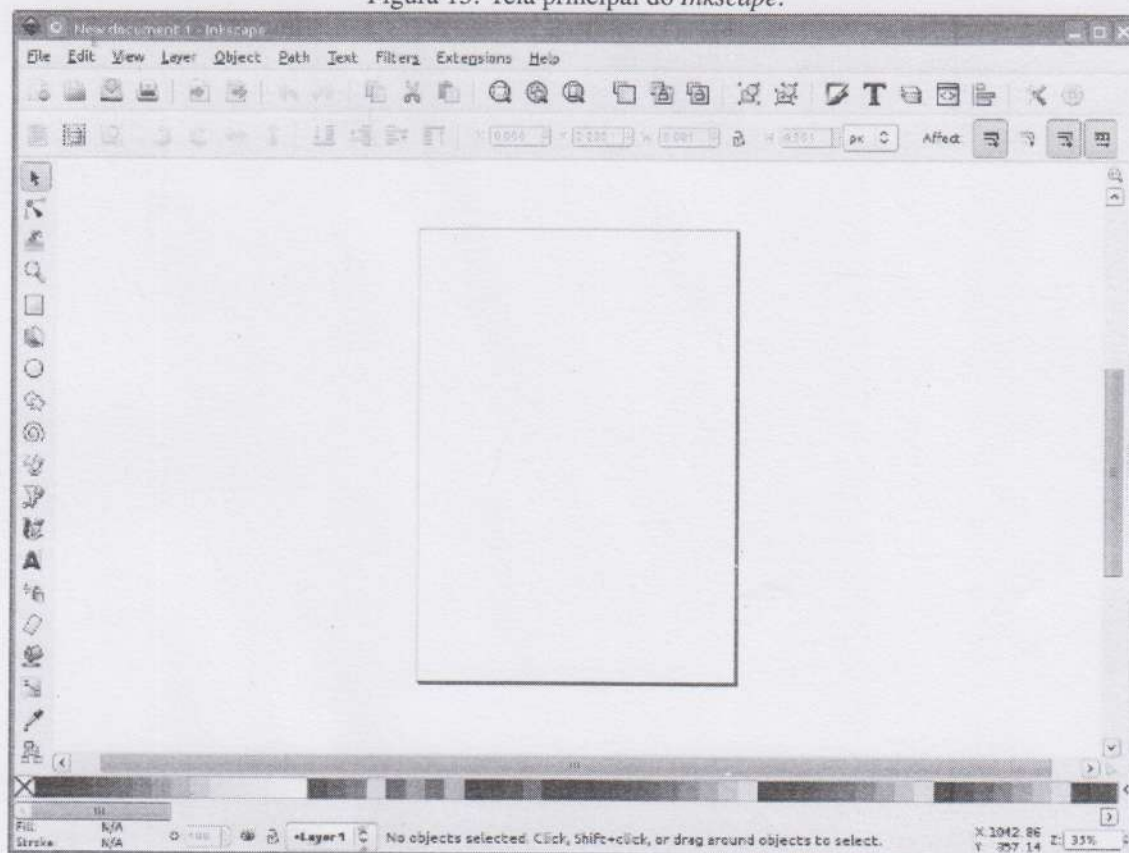
4.3.1 DEFINIÇÃO E HISTÓRICO

O site oficial do projeto o define como uma ferramenta open source que utiliza o padrão SVG como formato de arquivo padrão, além de XML e CSS2 (INKSCAPE, 2009a). É escrito em C/C++ e possui versões para as principais plataformas, foca seu desenvolvimento em um núcleo pequeno e na extensibilidade. Alguns dos recursos do SVG suportados são as formas básicas, caminhos, texto, clones, transformações gradientes e grupos. O *Inkscape* oferece suporte a edição de nós, camadas, metadados da licença *Creative Commons*¹⁹, importação e exportação em diferentes formatos, entre outros.

O histórico do projeto relata que o código inicial do projeto foi escrito em 1999 e ganhou o nome *Gill* (GNOME *Illustrator*). Em 2000 o projeto foi interrompido e deu lugar a uma

¹⁸<http://inkscape.org/>

¹⁹<http://www.creativecommons.org.br/>

Figura 15: Tela principal do *Inkscape*.

Fonte: Autoria própria.

nova versão baseada no código original, chamada *Sodipodi*, que teve maior popularidade, pois implementava novos recursos, suportava internacionalização, era multiplataforma e eliminava dependências (INKSCAPE, 2009b). O *Inkscape* surgiu em 2003 através de quatro desenvolvedores ativos do *Sodipodi*, que pretendiam focar o projeto no suporte total ao padrão SVG, em melhorias na interface e em uma abertura maior a contribuições da comunidade de desenvolvedores.

4.3.2 EXTENSÕES

Jurković e Scala (2011) destacam a flexibilidade e poder das ferramentas básicas do *Inkscape*, mas atentam para casos em que criar gráficos se torna repetitivo e trabalhoso, além de passível de erros. As extensões do *Inkscape* podem gerar gráficos complexos através da criação e manipulação de elementos e atributos, formatar textos e até mesmo realçar imagens rasterizadas. O *Inkscape* vem com várias extensões por padrão, mas é perfeitamente possível desenvolver novas extensões.

Jurković e Scala (2011) afirmam que as extensões do *Inkscape* são geralmente escritas em Python²⁰, mas é possível desenvolver extensões em qualquer linguagem que ofereça suporte à manipulação de dados XML, já que SVG é baseado em XML. Em relação à linguagem *Python*, é necessário instalar o módulo `python-lxml` para essa tarefa.

Um extensão é composta por um arquivo do tipo `*.inx` e um ou mais arquivos com o código na linguagem de programação escolhida. O arquivo `*.inx` é nada mais que um XML que define as características da extensão, como por exemplo o nome, o tipo, os campos e os tipo de dados que utilizará, onde será chamada na interface do *Inkscape*, entre outros. O código abaixo refere-se ao arquivo `*.inx` da extensão desenvolvida nesse trabalho:

```
<?xml version="1.0" encoding="UTF-8"?>
<inkscape-extension
xmlns="http://www.inkscape.org/namespace/inkscape/extension">
  <_name>Convert to canvas</_name>
  <id>org.inkscape.output.html5</id>
  <dependency type="executable" location="extensions">
    ink2canvas.py
  </dependency>
  <dependency type="executable" location="extensions">
    inkex.py
  </dependency>
  <output>
    <extension>.html</extension>
    <mimetype>text/html</mimetype>
    <_filename>HTML 5 (*.html)</_filename>
    <_filetypetooltip>HTML 5 canvas code</_filetypetooltip>
  </output>
  <script>
    <command reldir="extensions" interpreter="python">
      ink2canvas.py
    </command>
  </script>
</inkscape-extension>
```

O arquivo `inkex.py` é um módulo *Python* responsável por abstrair as operações comuns a todas as extensões, tratar e converter dados extraídos do SVG para estruturas de dados do *Python*, além de oferecer funções para facilitar o desenvolvimento. Há outros módulos como o `simplestyle.py` e `simplepath.py`, que possuem funções de manipulação de propriedades de estilo e caminhos, respectivamente.

²⁰<http://python.org>

4.4 TRABALHOS RELACIONADOS

4.4.1 CANVG

No universo de projetos de *software* livre há vários exemplos que apresentam similaridades mas possuem diferentes abordagens. É o caso da biblioteca *canvg*²¹, cuja função também é a conversão de SVG em código *Canvas*, escrita em *Javascript*. A meta do *canvg* é suportar completamente o SVG, inclusive animações. A biblioteca converte dinamicamente documentos SVG sob demanda, ou seja, não produz um arquivo, mas oferece a visualização do resultado.

A *canvg* permite a visualização de documentos SVG em dispositivos que não suportam esse formato, através da conversão instantânea para *Canvas*. O fato de se restringir ao contexto do navegador limita as possibilidades de sua aplicação, pois cria uma dependência de uma linguagem que é presente primariamente em navegadores. A ausência de uma interface gráfica dificulta a adoção por parte de usuários finais, já que a biblioteca é simplesmente um arquivo *Javascript* que precisa ser incluído em uma página HTML.

4.4.2 "FROM SVG TO CANVAS AND BACK" (KAIPAINEN; PAKSULA, 2010)

Nesse artigo, os autores descrevem um método experimental para renderizar SVG no *Canvas* através do suporte nativo do navegador à conversão de dados codificados, atentando para a segurança na transferência dos dados durante esse processo. Com isso definem o que seria uma aplicação de desenho baseado na *web* e as operações que deveriam ser suportadas nativamente por navegadores, o que contribuiria para a popularização do SVG e do *Canvas*:

- Desenho *bitmap*
- Desenho vetorial
- Interfaces
 - Exportar e importar para *bitmap* (do *Canvas* para o SVG)
 - Rasterização de vetores e importação (do SVG para o *Canvas*)
- Exportação
 - Exportar vetores
 - Exportar *bitmaps*

²¹<http://code.google.com/p/canvg/>

A proposta de exportação do SVG para o *Canvas* utiliza uma abordagem direta e simplista, porém limitada à visualização estática do resultado. Fazendo uso do processo de renderização do SVG pelo navegador à medida que é exibida na tela, é possível usar os pixels gerados diretamente no *Canvas*. Com isso é criado um objeto de imagem com dados serializados e codificados em *Base64*, que pode ser utilizado no *Canvas*. Além disso, os autores sugerem de forma semelhante o processo reverso, onde a imagem codificada é extraída do *Canvas* e incorporada ao SVG. Essa abordagem é útil para casos similares ao *canvg*, onde deseja-se converter documentos SVG dinâmica e instantaneamente para *Canvas*, mas devido à limitação de desse procedimento, que gera apenas a visualização, é impossível ter uma conversão dos dados de fato.

5 METODOLOGIA

Este trabalho constitui uma pesquisa e desenvolvimento de um *software* de conversão entre formatos gráficos abertos. A metodologia aplicada inicialmente na fase de concepção e pesquisa foi o levantamento bibliográfico. As especificações dos padrões *web* estudados foram analisadas em busca de informações úteis à compreensão do funcionamento e aplicação de seus recursos. A pesquisa de material técnico especializado permitiu complementar o estudo através de demonstrações e experimentos práticos.

Em seguida, foi realizada uma comparação dos padrões HTML5 e SVG em relação aos seus recursos, para definir a lógica de conversão e planejar o desenvolvimento inicial do projeto. A pesquisa e revisão constante das obras referenciadas permitiu orientar o desenvolvimento das fases posteriores do projeto. As fases foram relacionadas ao suporte gradual dos recursos e elementos especificados nos padrões estudados. Cada recurso foi testado e analisado com precisão, para garantir a conversão exata dos valores.

A etapa seguinte consistiu em estudar a estrutura de extensões do *software* de ilustração vetorial *Inkscape* para estabelecer a base inicial de código do projeto, no qual foi utilizada a linguagem *Python*¹.

Após a conclusão do *software*, foram realizados vários testes e os resultados foram comparados com os formatos originais, tanto visual como tecnicamente, expostos na forma de imagens. Essas imagens e comparações foram utilizadas como informação útil para as conclusões referentes ao trabalho.

¹<http://python.org/>

6 ANÁLISE DE RESULTADOS

Com a conclusão desse trabalho, vários dados foram levantados para permitir a análise do processo de desenvolvimento e dos resultados produzidos pela ferramenta desenvolvida, assim como a relação entre os seus fatores: os padrões e tecnologias utilizados.

6.1 SVG E HTML5

Durante a fase de pesquisa foi observado que os padrões SVG e HTML5 apresentam semelhanças em relação à nomenclatura de seus métodos, propriedades, aos recursos oferecidos, à ordem e tipos de parâmetros utilizados em elementos e métodos. Isso se deve ao fato de serem especificados pela mesma entidade, o W3C. Isso contribuiu para facilitar a compreensão dos formatos, e consequentemente, o processo de conversão. Há, no entanto, distinções a serem observadas entre os dois padrões. A compreensão da diferença entre imagens vetoriais e *bitmap* explicada na seção 4.2 é fundamental para a análise dessas distinções.

Em relação ao SVG, foi possível destacar as seguintes características:

- É um padrão baseado em modelo de árvore de objetos, similar ao HTML. Segue um modelo declarativo.
- Possui vários tipos de elementos que juntos compõem o desenho.
- A representação visual é criada por uma linguagem de marcação e formatação através de CSS e *scripts*.
- Permite adicionar eventos/interatividade diretamente aos seus objetos da mesma forma que no HTML, com *Javascript*.
- Por ser baseado em XML, o SVG é legível para seres humanos, logo é acessível, podendo ser “lido” por *softwares* sintetizadores de voz.

O SVG demonstrou ser mais adequado para aplicações interativas onde a qualidade dos gráficos tem mais importância que a eficiência de renderização. Gráficos vetoriais preservam os detalhes mesmo quando aumentados várias vezes, o que os torna desejados para impressões em alta resolução, geração de gráficos em servidores através de bancos de dados, simulações e aplicações que não exigem muitas modificações na estrutura de elementos, ou seja, que não comprometam a performance. Outros casos de uso são os mapas interativos, esquemas de engenharia, gráficos de dados e fluxogramas.

O SVG é uma boa escolha para esses casos porque é fácil gerar gráficos a partir de dados em XML ou em outros formatos de texto e em aplicações que necessitam de precisão visual. O SVG pode ser utilizado em editores vetoriais, como é o caso do *Inkscape*. A interação com os usuários e a acessibilidade são pontos fortes. O SVG também pode ser usado como imagens estáticas em sites e ícones de aplicações, substituindo as imagens comuns, garantindo a qualidade em todos os casos de uso, já que pode ser visualizado em navegadores. Em contraste com o SVG, foram observadas as seguintes características no *Canvas*:

- É baseado em *pixels*, ou seja, o *Canvas* é essencialmente um elemento de imagem com uma API de desenho. Segue um modelo procedural.
- É um elemento único, similar ao `<img/>`.
- Toda a representação gráfica é feita através de *scripts*.
- Toda a interatividade do *Canvas* deve ser programada manualmente. Como não possui elementos, as ações dependem do conhecimento da posição do *mouse* ou de ações do teclado.
- A API do *Canvas* ainda não provê acessibilidade. O único meio é fornecer conteúdo alternativo em HTML.

O fato de trabalhar em um nível mais baixo em relação ao SVG permite ao *Canvas* trabalhar mais eficientemente com aplicações em tempo real que exigem um elevado processamento gráfico, como jogos e simulações animadas. Apesar disso, deve-se recorrer a uma análise mais precisa sobre a escolha da tecnologia a ser utilizada, pois tudo depende da aplicação a ser desenvolvida. Por ter a capacidade de manipular *pixels*, o *Canvas* pode ser utilizado para edição de imagens *on-line*, com aplicação de filtros e efeitos dinâmicos, o que é um fator diferencial em relação ao SVG.

O fato do *Canvas* ser uma superfície de desenho, tal qual um quadro branco, torna-o uma escolha certa para aplicações de visualização de dados massivos, como mapas e gráficos

dinâmicos. No SVG esse exemplo seria impraticável devido à grande quantidade de elementos que deveriam ser manipulados, o que reduziria a performance da aplicação.

Há casos, porém, em que ambas as tecnologias podem ser usadas em conjunto: mapas, jogos e demais aplicações interativas, explorando as vantagens de cada uma. O fato de serem similares em alguns pontos ajuda nesse sentido. Um exemplo pode ser utilizar SVG para interação com o usuário e animações simples com Javascript, enquanto o Canvas se encarrega de produzir efeitos especiais e animações.

6.2 O INKSCAPE E O SVG

Durante o desenvolvimento da extensão foi observado que o *Inkscape*, apesar de ainda não implementar 100% o padrão SVG, suporta normalmente elementos básicos como `<circle>` e `<line>`. Porém, ao utilizar as ferramentas do *Inkscape* para desenhar essas formas e outras mais complexas como estrelas e espirais, ele cria elementos do tipo `<path>` para representá-las e adiciona o atributo `sodipodi:type` com o tipo de objeto relacionado (i.e.: estrela, círculo, etc). Isso se deve ao fato do SVG não possuir elementos que representem algumas ferramentas do *Inkscape*, como estrelas, espirais e cubos 3D.

Quanto aos círculos e elipses, suas especificações no SVG não permitem implementar os recursos de arcos e segmentos que o *Inkscape* oferece, como ilustrado na figura 16. Por isso também são desenhados com o elemento `<path>` e o atributo `sodipodi:type`. O mesmo acontece com os elementos `<line>` e `<polyline>`, pois não permitem desenhar curvas. Essa questão não se aplica aos retângulos. O elemento `<rect>` é usado normalmente, pois oferece todos os recursos que a ferramenta Retângulo do *Inkscape* precisa, como por exemplo, bordas arredondadas. Essa questão foi levada em conta ao se desenvolver a extensão, implementando os mesmos elementos SVG que o *Inkscape*.

6.3 DESENVOLVIMENTO E PROCESSO DE CONVERSÃO

A *Ink2canvas* foi desenvolvida em *Python*, que é uma linguagem bastante versátil e fácil de trabalhar. O código produzido é legível e fácil de manter, facilitando as constantes refatorações no projeto. Todo o código-fonte foi gerenciado através do sistema de controle de versão *git*¹ e mantido pelo serviço de repositório virtual *GitHub*². A extensão foi desenvolvida como um *software* livre e permite a colaboração de desenvolvedores de todo o mundo. Mesmo que ainda

¹<http://git-scm.com/>

²<https://github.com/karlisson/ink2canvas>

não tenha atingido todas as metas planejadas, o código já se encontra no repositório oficial do projeto do *Inkscape*, em status de desenvolvimento.

O SVG é representado como uma árvore de elementos, armazenada em memória e acessível através de operações de matrizes. A extensão extrai os dados através da varredura dessa árvore em busca dos elementos que devem ser renderizados. Esses dados são tratados e enviados como parâmetros para métodos que geram os comandos equivalentes do *Canvas*. Ao final desse processo, um arquivo *.html é criado com todo o código produzido pela extensão. O trecho de código abaixo exhibe um elemento SVG e o código *Canvas* gerado a partir da extensão:

```
//Código SVG original
<path style="stroke:#000" d="M 70,85 L 217,243"/>

//Código Canvas gerado
ctx.beginPath();
ctx.lineJoin = 'miter';
ctx.strokeStyle = 'rgb(0, 0, 0)';
ctx.lineCap = 'butt';
ctx.lineWidth = 1;
ctx.moveTo(70, 85);
ctx.lineTo(217, 243);
ctx.stroke();
```

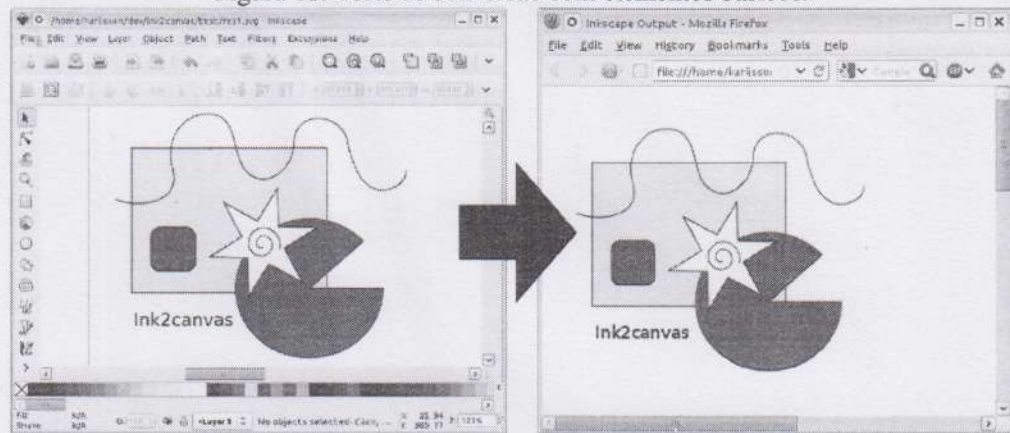
6.4 TESTES REALIZADOS

A *Ink2canvas* apresentou uma performance aceitável desde o início de seu desenvolvimento. Vários testes foram realizados, apresentando bons tempos de execução entre 80 a 130 milissegundos para desenhos simples, utilizando elementos básicos do SVG. Para cada teste realizado no *Inkscape* um arquivo HTML foi criado e visualizado no navegador utilizado, o *Firefox* 8.0. Em seguida o código gerado foi analisado.

Devido ao modo inicial como foi implementado, o código *Javascript* gerado pela extensão apresenta redundâncias. Atributos de estilo são definidos em todos os objetos convertidos. No caso de haver dois objetos em sequência que possuam a mesma cor de preenchimento, o atributo *context.fillStyle* será repetido com o mesmo valor, o que gera repetição de código e contribui para o aumento do tamanho do arquivo. A otimização necessária para esse caso envolve definir o atributo de preenchimento apenas no primeiro objeto, e alterá-lo apenas quando uma nova cor de preenchimento for necessária.

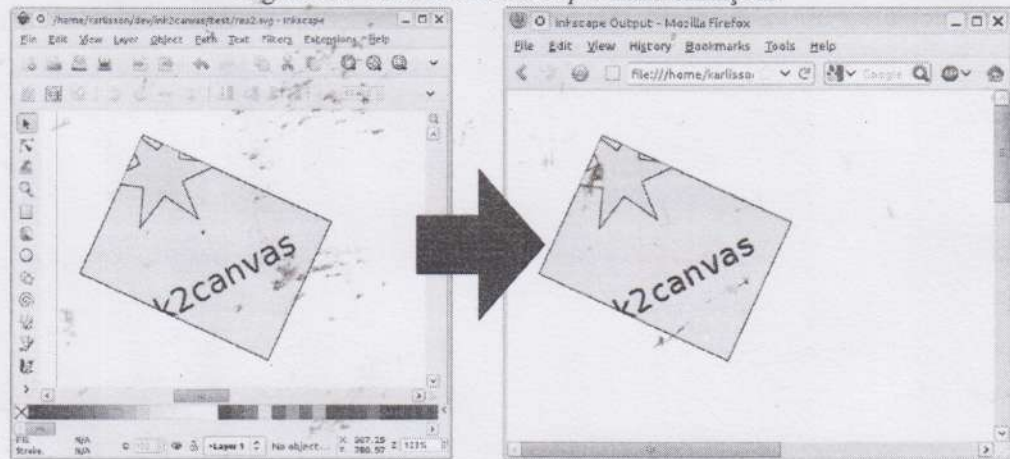
A figura 16 apresenta o resultado do teste com formas básicas como caminhos, retângulos, estrelas, espirais e círculos.

Figura 16: Teste de conversão com elementos básicos.



Fonte: Autoria própria.

A figura 17 ilustra operações mais complexas como área de *clip*, texto e transformações. No exemplo, um elemento de texto e um polígono foram “clipados” pelo retângulo. As rotações e translações aplicadas também foram convertidas normalmente. Percebe-se nessa imagem gerada, assim como na figura 16, que a renderização de texto no *Canvas* ainda não se comporta como esperado, apresentando significantes disparidades em relação ao original.

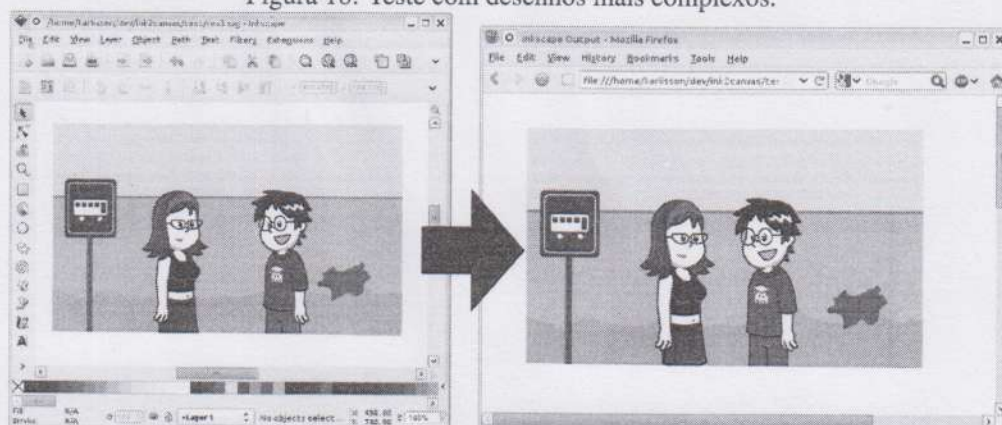
Figura 17: Teste de área de *clip* e transformações.

Fonte: Autoria própria.

Foi realizado um teste com um arquivo mais complexo que os anteriores, como visto na figura 18. Apesar de composto de inúmeros objetos com diferentes propriedades, o arquivo foi convertido de acordo com o resultado esperado. Nesse teste foram contabilizados 180 milissegundos de tempo de execução, demonstrando a eficácia da extensão para arquivos SVG mais complexos.

A *Ink2Canvas* foi desenvolvida com suporte básico a elementos de texto, mas ainda não

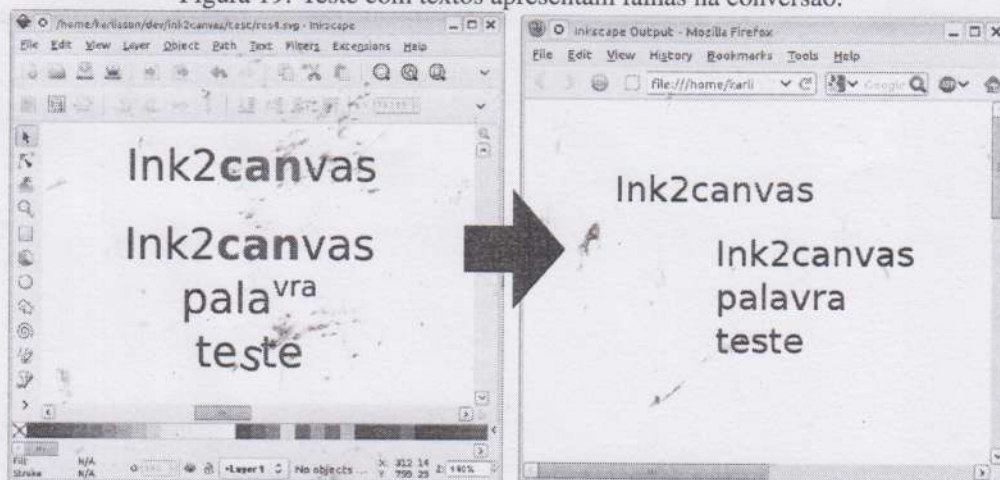
Figura 18: Teste com desenhos mais complexos.



Fonte: Autoria própria.

implementa certas propriedades de texto como alinhamento, efeito de sobrescrito e formatação de caracteres individuais, conforme visto na figura 19. A formatação de caracteres ou trechos de um elemento de texto é particularmente complexa pois se baseia em uma estrutura de elementos do tipo `<tspan>` aninhados, onde se faz necessária recursão para ler todos os elementos.

Figura 19: Teste com textos apresentam falhas na conversão.



Fonte: Autoria própria.

7 CONCLUSÃO

A partir das informações obtidas na pesquisa inicial foi possível realizar uma análise dos padrões *web* SVG e HTML5 em relação às suas características, vantagens, desvantagens e casos de uso. O conhecimento adquirido nesse estudo foi de fundamental importância para o desenvolvimento da extensão *Ink2canvas*, que apresentou resultados satisfatórios e permitiu chegar às conclusões expostas a seguir.

O HTML5 e o SVG são considerados tecnologias modernas e eficazes. A adoção de ambos cresce à medida que as aplicações *web* tornam-se cada vez mais interativas e dinâmicas. O ponto forte do SVG reside na interatividade, dado que seus elementos podem disparar eventos de acordo com o tipo de ação do usuário. O formato aberto e legível é uma grande vantagem em relação à interoperabilidade e à acessibilidade, o que facilita a conversão para outros tipos de dados e utilização em conjunto com outros padrões como o HTML. A escalabilidade inerente às imagens vetoriais fazem do SVG uma escolha certa onde qualidade e precisão são fatores imprescindíveis. Criar código SVG manualmente é um esforço dispendioso, mas editores SVG automatizam essa tarefa. A proposta do *Inkscape* é oferecer total suporte à especificação do SVG, prover uma interface simples e completa, permitindo também adicionar propriedades, eventos e comandos em *Javascript*. Dadas essas características, o SVG é uma excelente escolha como formato de arquivo para um editor vetorial livre como o *Inkscape*.

O HTML5 ainda está em processo de especificação, mas já é implementado nos navegadores modernos e utilizado em vários projetos, alguns experimentais. Considera-se atualmente que o termo HTML5 refere-se a um conjunto de tecnologias *web*. Um desses componentes, o elemento *Canvas*, é um dos recursos mais esperados nessa nova versão do HTML. Enquanto o SVG trabalha com gráficos vetoriais, o *Canvas* opera a nível de *pixels*. Esse é o grande diferencial dessa tecnologia e ao mesmo tempo uma desvantagem. As operações gráficas de baixo nível são extremamente rápidas. Essa vantagem tem sido explorada em editores de imagens *on-line*, em jogos e animações, apresentando resultados melhores que o SVG nesse sentido. Porém, a performance pode ser facilmente comprometida por diferentes fatores, como o *hardware*, o navegador ou detalhes da implementação (e.g., redesenhar toda a área do *Canvas* a cada

atualização de quadro em um jogo é um processo custoso. Recomenda-se redesenhar apenas a área onde ocorreu a modificação de cena).

Assim, verifica-se que o uso do *Canvas* é indicado em casos onde a performance é o fator principal, como em jogos e aplicações em tempo real ou que trabalham com grandes quantidades de dados. O SVG é mais adequado para simulações, gráficos e aplicações interativas, que independam da resolução da tela do usuário. Conclui-se então que a escolha da tecnologia ideal depende do tipo de aplicação a ser desenvolvida, sendo possível ainda empregar ambas em uma mesma aplicação, combinando suas vantagens.

A API do *Canvas*, assim como o SVG, sugere o uso de ferramentas para automatizar o trabalho. A *Ink2canvas* foi desenvolvida com esse objetivo, permitindo gerar código *Canvas* através do *Inkscape*. Durante a fase de implementação foram encontradas dificuldades para abstrair os diferentes tipos de elementos do SVG, por conterem diferentes tipos de dados e parâmetros. A cada novo elemento suportado pela extensão foi preciso reorganizar o código para eliminar repetições e modularizá-lo. Todos os elementos básicos foram adicionados quando os primeiros testes foram realizados, comprovando a eficácia da conversão.

Dados os objetivos e o contexto da pesquisa realizada, pode-se concluir, portanto, que a extensão *Ink2canvas* não apenas demonstrou ser possível gerar gráficos HTML5 *Canvas* a partir do SVG, como atingiu seu objetivo principal, que é proporcionar ao *Inkscape* essa capacidade, contribuindo para torná-lo um *software* de ilustração vetorial completo e atualizado em relação às novas tecnologias *web*.

7.1 TRABALHOS FUTUROS

O código *Canvas* gerado pela extensão é composto de valores numéricos absolutos, relativos à sua posição no espaço definido pela área de visualização. Este projeto pode ser ampliado de forma a **parametrizar** os comandos *Canvas* gerados. Para isso seria necessário encapsular trechos do código relativos a elementos distintos e tratá-los como instâncias de uma classe de representação visual genérica, com atributos como posição, cores e transformações. Essa melhoria seria útil, por exemplo, para a manipulação de personagens de jogos, ou partes deles, diretamente no código *Javascript*, eliminando a necessidade de alteração de propriedades simples no *Inkscape*.

REFERÊNCIAS

- EISENBERG, J. David. *SVG Essentials: Producing Scalable Vector Graphics with XML*. 1. ed. Sebastopol: O'Reilly Media Inc, 2002. ISBN 0-596-00223-8.
- FLANAGAN, David. *Canvas Pocket Reference: Scripted Graphics for HTML5*. 1. ed. Sebastopol: O'Reilly Media Inc, 2010. ISBN 978-1-449-39680-0.
- FULTON, Steve; FULTON, Jeff. *HTML5 Canvas: Native Interactivity and Animation for the Web*. 1. ed. Sebastopol: O'Reilly Media Inc, 2011. ISBN 978-1-449-39390-8.
- INKSCAPE. *About Inkscape*. 2009. Disponível em: http://wiki.inkscape.org/wiki/index.php/About_Inkscape>. Acesso em: 26 nov. 2011.
- INKSCAPE. *Inkscape History*. 2009. Disponível em: <http://wiki.inkscape.org/wiki/index.php/InkscapeHistory>>. Acesso em: 26 nov. 2011.
- JURKOVIĆ, Mihaela; SCALA, Rigel Di. *Inkscape 0.48 Illustrator's Cookbook*. Birmingham: Packt Publishing, 2011. ISBN 978-1-849512-66-4.
- KAIPAINEN, Samuli; PAKSULA, Matti. *From SVG to Canvas and Back*. 2010. Disponível em: http://www.svgopen.org/2010/papers/62-From_SVG_to_Canvas_and_Back/>. Acesso em: 30 nov. 2011.
- PILGRIM, Mark. *HTML5 UP and Running: Dive into the Future of Web Development*. 1. ed. Sebastopol: O'Reilly Media Inc, 2010. ISBN 978-0-596-80602-6.
- W3C. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. 2011. Disponível em: <http://www.w3.org/TR/SVG/>>. Acesso em: 06 set. 2011.
- WHATWG. *HTML5 - Living Standard*. 2011. Disponível em: <http://www.whatwg.org/html>>. Acesso em: 20 set. 2011.