Curso de Sistemas de Informação
Universidade Estadual de Mato Grosso do Sul

Aplicações WEB 3D com WEBGL: Visualizador de Malhas

Allan Henrique Paza

Msc. Diogo Fernando Trevisan
(Orientador)

Dourados - MS
2015

Curso de Sistemas de Informação
Universidade Estadual de Mato Grosso do Sul

Aplicações WEB 3D com WEBGL: Visualizador de Malhas

Allan Henrique Paza

Novembro de 2015

Banca Examinadora:

Prof. MSc. Diogo Fernando Trevisan (Orientador)
Sistemas de Informação - UEMS

Prof. Alcione Ferreira
Sistemas de Informação - UEMS

Profa. Dra. Mercedes Rocío Gonzales Márquez
Sistemas de Informação - UEMS

Aplicações WEB 3D com WEBGL: Visualizador de Malhas

Allan Henrique Paza

Este exemplar corresponde à redação final da monografia da disciplina Projeto Final de Curso devidamente corrigida e defendida por Allan Henrique Paza e aprovada pela Banca Examinadora, como parte dos requisitos para a obtenção do título de Bacharel em Sistemas de Informação.

Dourados, 23 de novembro de 2015.

Prof Msc. Diogo Fernando Trevisan
(Orientador)

Resumo

Hoje em dia a internet nos proporciona informação e diversão. Permite também fazer compras, estabelecer contato com parentes e amigos e conhecer lugares turísticos. Porém ao visualizar um produto numa loja virtual ou ver uma foto (ou vídeo) de uma paisagem estamos limitados à perspectiva e ao ângulo de quem fotografou (ou filmou).

A vantagem de utilizar a modelagem 3D é que o usuário tem a liberdade de visualizar seu objeto no ângulo que ele quer, pode mudar a cor e aumentar o zoom sem perder a qualidade da imagem.

A WebGL tem o propósito de juntar os benefícios da internet com os da modelagem 3D. Este trabalho apresenta a WebGL, assim como seus requisitos e suas vantagens. E para demonstrar seu poder e seu uso foi implementado um visualizador de malhas para *web*.

Palavras-chave: WebGL, modelagem web 3D, HTML5, canvas, visualizador de malhas.

Abstract

Nowadays the internet provides us with information and fun. Also allows us to shop, stay in touch with relatives and friends and visit tourist places. However when viewing a product in a virtual store or see a photo (or video) of a landscape we are limited to the perspective and angle of whom photographed (or filmed).

The advantage of using 3D modeling is that the user has the freedom to view your object from the angle he wants, can change color and zoom without losing image quality.

The WebGL has the purpose to bring together the benefits of the internet with 3D modeling. This work presents the WebGL as well as its requirements and its advantages. And to demonstrate its power and its use was implemented a mesh viewer for web.

Keywords: WebGL, 3D web modeling, HTML5, canvas, mesh viewer.

Conteúdo

Lista de Siglas

OpenGL *Open Graphics Library*

API *Application Programming Interface*

GPU *Graphics Processing Unit*

HTML *Hyper Text Markup Language*

CSS *Cascading Style Sheets*

XML *eXtensible Markup Language*

DOM *Document Object Model*

GLSL *OpenGL Shading Language*

iOS *iPhone OS*

VBO *Vertex Buffer Object*

VAO *Vertex Array Object*

HTTP *Hypertext Transfer Protocol*

PHP *Hypertext Preprocessor*

2D Duas Dimensões

3D Três dimensões

WAMP Windows, Apache, MySQL e PHP/Perl/Python

LAMP Linux, Apache, MySQL e PHP/Perl/Python

Lista de Figuras

Listagens

1 Introdução

1.1 Introdução

Quando, na década de 90, Sir Tim Berners-Lee inventou a *web*, ele não imaginava que ela se tornaria o principal ingrediente da globalização das comunicações. A *web* hoje é referência em comércio, estudo e informação, pois qualquer notícia está instantaneamente divulgada para o mundo todo. Sobre o comércio, qualquer um de qualquer lugar do globo consegue comprar o que quiser de qualquer país. Na educação, é cada vez mais comum cursos à distância, tanto de formação acadêmica quanto profissional (treinamento de empresas) [24].

Na década de 90 também surgiu o OpenGL, desenvolvido na *Silicon Graphics Computer Systems*. Essa é uma API[1] utilizada para a criação de aplicações gráficas que podem acessar recursos da placa gráfica (GPU[2]) [23].

A WebGL vem ser a junção dessas duas tecnologias, *web* e OpenGL. A *web* é referência em informação: imagine que ao buscar informações sobre um filme pudesse viajar pela história do mesmo, em 3D.[3] Ou buscar informações sobre uma catedral e pudesse conhecê-la por dentro, como se estivesse andando dentro dela. Se o comércio *web* se tornou uma prática comum, imagine se para escolher um produto você pudesse ver não apenas fotos, mas sim em

[1] *Application Programming Interface* - Interface de Programação de Aplicativos
[2] *Graphics Processing Unit* - Unidade de Processamento Gráfico
[3] http://middle-earth.thehobbit.com/ - O *Hobbit*: A Desolação de *Smaug* - Viagem pela Terra Média - Acessado em 07/10/2014 as 15:20

3D, girar e ver em todos os ângulos. E se ao estudar via *web* anatomia tivesse à disposição todos os sistemas do corpo humano também em 3D. Imaginou? Isso está virando realidade devido à WebGL.

A WebGL é uma API relativamente nova e em pleno desenvolvimento. Como o uso de aplicativos tridimensionais pode trazer mais imersão à *web*, avaliou-se as principais dificuldades e características da WebGL. Atualmente é comum aplicações *web* fornecerem muita interatividade, diferente do HTML estático de antigamente. Neste trabalho também foi verificado se a WebGL está preparada para ser utilizada para a rápida criação de aplicações *web* em 3D.

1.2 Objetivos

Este projeto tem como objetivo realizar um estudo sobre a WebGL e implementar um visualizador de malhas para *web*. Para isso, serão feitos diversos estudos de caso abordando:

- Criação de primitivas;
- Transformações geométricas (translação, rotação e escala);
- Tonalização e iluminação;
- Estruturas de dados (*Halfedge*)
- Projeções e controle de câmera.
- *Shaders*

Processadores de malhas são úteis para realizar operações como remoção de vértices não referenciados, faces com área nula além de simplesmente permitir a visualização de malhas. Um software similar é o Meshlab [4].

Neste trabalho foi implementado um processador de malhas *web* e espera-se que o estudo sobre a API WebGL sirva como ponto de partida para outros acadêmicos que queiram estudar a API.

[4] http://meshlab.sourceforge.net/

1.3 Organização do Texto

No capítulo 2 será apresentada a revisão bibliográfica. Antes de começar a citar WebGL, é necessário falar dos componentes que tornaram o seu uso possível. É o caso da HTML5, que com ela surgiu o elemento canvas que cede um espaço no navegador para que a WebGL, através do JavaScript, consiga desenhar na tela. Após citada a HTML5 e suas mudanças em relação a HTML4, há uma introdução do JavaScript. Essa introdução mostra a relação do canvas e do JavaScript. Em seguida, no capítulo 3, é citado a WebGL, com uma breve introdução, as vantagens, versões, uso e algumas informações técnicas. Posteriormente, o capítulo 4 mostra como é o visualizador e o que é necessário para o seu funcionamento. O aplicativo de visualização de malhas via *web* é apresentado no capítulo 5. E por fim são apresentados os resultados deste projeto e o que se espera da WebGL.

2 HTML 5

2.1 Introdução

HTML, do inglês *Hypertext Markup Language*, é a linguagem que fornece a estrutura da página da *web*, onde o autor pode publicar documentos, tabelas, fotos, vídeos, etc. A HTML, resumidamente, como o próprio nome sugere, é feita no conceito de hipertextos e tem como atributo o intuito de interligar os documentos na *web* [24].

A HTML foi criada por Tim Berner-Lee no início da década de 90. Foram lançadas as versões HTML, HTML +, HTML 2.0, HTML 3.0, e havia ainda implementações proprietárias, porém apenas em 1997 que a versão padronizada foi lançada, a versão HTML 3.2. Logo após foram lançadas as versões HTML4 e HTML 4.01 como recomendações oficiais [24].

Neste capítulo, teremos um breve relato sobre a HTML5, que é a última versão do padrão HTML, e a descrição do elemento canvas, o qual é utilizado para geração de páginas 3D com WebGL.

2.2 Principais mudanças em relação ao HTML 4

A HTML5 tem o objetivo de substituir a HTML4 e evitar o uso de *plug-ins* adicionais. Ela também facilita o uso do CSS[1] e do JavaScript, evitando longos códigos adicionais e o desenvolvimento deve ser nítido para o usuário final. Segundo Silva [24] (2011): "As principais diferenças entre a HTML5 e a HTML4 têm suas origens no fato de a HTML5 estar sendo desenvolvida com o propósito de substituir tanto a HTML criada no anos 90 quanto a XHTML que foi uma tentativa frustrada de reformular a HTML4 como aplicação XML".

Uma das mudanças em relação à versão 4 é a mudança do estado de "em desuso" de alguns elementos para "obsolet". Na nova versão não existem elementos em desuso, eles apenas não são recomendados, ou seja, na prática nenhum site precisa ser refeito para se adequar às novas regras.

A sintaxe do *DOCTYPE*, por exemplo, foi bem simplificada, necessitando apenas de '<!DOCTYPE html>'; Há novos elementos para manipulação de conteúdo (<article>, <footer>, <header>, <nav>, <section>) para criações de seções, cabeçalhos e rodapés, elementos de reprodução de mídia (<audio> e <video>). Outro recurso novo da HTML5 é o suporte para armazenamento local (*localStorage*) cuja finalidade é semelhante ao dos *cookies*, porém ele armazena dados persistentes de várias janelas ou abas do navegador a fim de compartilhar dados entre si. E também o elemento <canvas>, o qual será o foco desse projeto e diversos outros não relevantes neste momento [24].

2.2.1 Elemento canvas

O elemento canvas foi criado com o objetivo de reservar um espaço na página destinado à criação dinâmica de imagens ou animações em 2D ou 3D[2]. Na Listagem 2.1 pode-se ver o uso da *tag* canvas, com dimensão de 200x100

[1] *Cascading Style Sheets*

[2] A HTML5 não possui o recurso pra 3D, por isso usa-se a WebGL.

pixels.

```
<canvas id="meuCanvas" width="200" height="100"></canvas>
```

Listagem 2.1: Criando o elemento canvas.

O elemento canvas reserva o espaço e o resto é feito através do JavaScript, conforme é mostrado na Listagem 2.2.

```
<script>
var c = document.getElementById("meuCanvas");
var ctx = c.getContext("2d");
ctx.fillStyle = "#0000FF";
ctx.fillRect(0,0,150,75);
</script>
```

Listagem 2.2: Exemplo de canvas.

O resultado será um retângulo azul um pouco menor do que reservado pelo canvas anteriormente, como é mostrado na Figura 2.1.

Figura 2.1: Exemplo de canvas.

2.3 JavaScript

JavaScript é uma das mais populares linguagens de programação do mundo. Ela é uma linguagem de *script* orientada a objetos, pequena, leve e multi-plataforma utilizada geralmente em navegadores *web*.

2.3.1 Visão Geral

Geralmente quando se fala de JavaScript fala-se também do *Document Object Model* - DOM. No DOM há itens de nível superior, itens aninhados abaixo dele e itens agrupados em blocos. Esses itens são chamados de nós e

cada nó tem relação com seus nós adjacentes. Portanto o DOM é um modelo de mapeamento de *layout* para o HTML fazendo com que o JavaScript consiga acessar o nó e fazer o que se deseja [32].

Um dos métodos mais usados do JavaScript para acessar um nó do DOM é o document.getElementById(), o qual poderemos utilizar para mudar, deletar, criar, copiar elementos HTML entre muitas outras funções. Para exemplificar será utilizado o JavaScript para mudar a cor e o tamanho do parágrafo (<p>). <p id="paragrafo">Esse é um parágrafo qualquer.< /p> A função que modificará o parágrafo é declarada da forma apresentada na Listagem 2.3.

```
<script>
  function minhaFuncao() {
    var x = document.getElementById("paragrafo");
    x.style.fontSize = "25px";
    x.style.color = "red";
  }
</script>
```

Listagem 2.3: Código JavaScript.

Na primeira e última linha tem-se *tags* indicando que o código dentro é um código JavaScript. A segunda linha indica a criação de uma função e a nomeia como minhaFuncao. Na terceira é criada uma referência da variável x com o elemento com o id "paragrafo", no caso é um elemento (*tag*) <p> do HTML que define um paragrafo. Na quarta e quinta linha altera-se o CSS do elemento <p>, o tamanho da fonte e a cor respectivamente.

2.3.2 JavaScript e canvas

Como foi visto no item 2.2.1 o elemento canvas do HTML precisa de um script para apresentar o resultado esperado. Da mesma forma mostrada na Listagem 2.3 onde foi utilizado o JavaScript para alterar o CSS de um elemento <p> do HTML, este será utilizado para a manipulação do elemento <canvas>.

Para exemplificar: primeiramente é criado o elemento <canvas>, como é mostrado na Listagem 2.4.

```
<canvas id="meuCanvas" width="200" height="100" style="border:2
    px solid #d3d3d3;">
  Seu navegador não suporta o elemento canvas!
</canvas>
```

Listagem 2.4: Exemplo JavaScript e canvas.

O resultado pode ser visto na Figura 2.2. Ele apenas reserva um espaço na tela do navegador com as dimensões de 200x100 *pixels*.

Figura 2.2: Criação de um elemento canvas.

O texto na segunda linha apenas aparecerá quando o navegador não suportar o HTML 5 e consequentemente o elemento <canvas>. Após a declaração do elemento <canvas> será declarado o script como mostra a Listagem 2.5.

```
<script>
var c=document.getElementById("meuCanvas");
var ctx=c.getContext("2d");
ctx.moveTo(0,0);
ctx.lineTo(200,100);
ctx.stroke();
</script>
```

Listagem 2.5: Exemplo JavaScript e canvas.

Na terceira linha é utilizado o método getContext() existente na API e canvas para criar a referência entre a variável ctx ao contexto de criação gráfica. Essa referência é fundamental para que possam ser acessados os métodos e atributos JavaScript da API. O exemplo utilizado é bidimensional, portanto o elemento canvas possui duas dimensões x e y, segundo um sistema de coordenadas cuja origem é no canto superior esquerdo, como mostra a Figura 2.3.

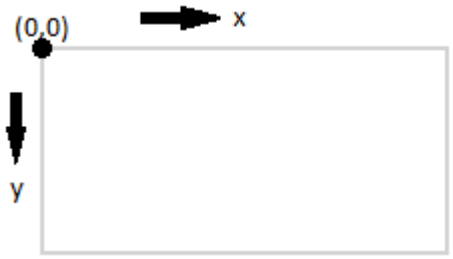

Figura 2.3: Exemplo de canvas bidimensional.

No exemplo é criada uma linha do ponto inicial 0,0 (linha 4) até o final 200,100 (linha 5) conforme pode ser visto na figura 2.4.

Figura 2.4: Exemplo de linha no canvas bidimensional.

3 WebGL

3.1 Introdução

A WebGL é um padrão *web* gratuito que começou a surgir em 2006 quando Vladimar Vukićević trabalhou no protótipo do canvas 3D usando OpenGL para *web*. Em 2009, Khronos Group criou o *WebGL Working Group*. O contexto 3D foi modificado para WebGL, e a versão 1.0 foi completa em 2011 [7].

WebGL é um padrão *web* gráfico, de baixo nível, livre de royalties, multiplataforma e baseada em OpenGL ES[1] 2.0. Essa utiliza a placa gráfica (GPU) e o conteúdo 3D é mostrado ao usuário através do elemento canvas do HTML5. Os desenvolvedores familiarizados com OpenGL ES 2.0 reconhecerão a WebGL como uma API baseada em *Shader*[2] usando GLSL, com construções que são semanticamente semelhantes aos do subjacente OpenGL ES 2.0 API. [11].

Existem seis versões da WebGL, cujo quatro delas são estáveis: 1.0.0, 1.0.1, 1.0.2 e 1.0.3 [29]; e duas estão na fase beta de desenvolvimento: 1.0.4 e 2.0.0 [28].

Grandes fabricantes de navegadores fazem parte do grupo que mantem a WebGL:

[1] *Embedded Systems* - Sistemas Embarcados

[2] Um *shader* é um pedaço de código de programa que implementa algoritmos para obter os *pixels* de uma malha para a tela [17].

- Google (Chrome)
- Opera (Opera)
- Mozilla (Firefox)
- Apple (Safari)

Os fabricantes dos navegadores citados acima têm grande papel para desenvolver e apoiar a WebGL, e seus engenheiros são membros-chave do grupo de trabalho que desenvolve a especificação. O processo de especificação da WebGL é aberto a todos os membros da Khronos, e há também listas de discussão abertas ao público.

3.2 Principais Vantagens

A WebGL oferece uma série de vantagens porque ela é baseada em OpenGL e será integrada em todos os navegadores populares, além de funcionar em várias plataformas inclusive nas móveis [16]. Pode-se citar como vantagens:

- Uma API que é baseada em um padrão familiar e amplamente aceita de gráficos 3D, o OpenGL.
- Compatibilidade multi-navegador, já que os maiores fabricantes de navegadores fazem parte do grupo que mantem a WebGL.
- É multi-plataforma, ou seja, funciona nos principais sistemas operacionais tanto de computadores pessoais quanto móveis.
- A forte integração com o conteúdo HTML, incluindo composição em camadas, a interação com outros elementos HTML e uso de mecanismos de manipulação de eventos.
- Gráficos 3D acelerados por hardware para o ambiente do navegador, ou seja, a WebGL utiliza a placa de vídeo do computador para processar os gráficos.

- Um ambiente de *script* que torna mais fácil criar protótipos de gráficos 3D, visualizar e depurar os gráficos renderizados sem a necessidade de compilar o código antes.

- Gerenciamento automático de memória: Ao contrário de seu primo OpenGL e outras tecnologias em que há operações específicas para alocar e desalocar memória manualmente, a WebGL não tem esse requisito. Ele segue as regras para escopo de variáveis em JavaScript e a memória é desalocada automaticamente quando não é mais necessária.

- WebGL é suportado em quase todos os navegadores móveis: Chrome mobile (Android), Firefox móvel (Android e Firefox OS), Amazon Silk (Kindle Fire HDX), Tizen (novo sistema operacional da Intel) e BlackBerry 10.

- O desempenho de aplicações WebGL é comparável às aplicações *standalone*. Isso ocorre porque a WebGL utiliza a placa de gráfica local [4].

3.3 Pré-requisitos para programar em WebGL

Todo programador que deseja se aventurar na programação 3D *web* e utilizar a WebGL deverá conhecer a GLSL (*OpenGL Shading Language*), a linguagem de *Shader* utilizada pelo OpenGL. A WebGL é uma API de baixo nível, ou seja, mesmo as coisas simples em WebGL requerem um pouco de código. Cálculo de matrizes é utilizado para configurar transformações e os *Buffers* de vértice para armazenar dados sobre as posições dos vértices, normais, cores e texturas [12].

3.4 Implementação usando navegador

A especificação WebGL 1.0 foi lançada recentemente, e as últimas versões de vários navegadores estão perto de alcançar total conformidade. Não neces-

sita de quaisquer etapas manuais para habilitá-lo, ou seja, apenas a instalação da última versão do navegador [10].

Como foi citado anteriormente, as quatro grandes fabricantes de navegadores fazem parte do grupo da WebGL, assim, serão utilizados esses navegadores nos exemplos para a implementação.

No Firefox a WebGL é suportada a partir da versão 4.0. Para fins de teste, depuração e processamento de software pode ser usado através de OSMesa (off-screen Mesa)[3] ou também habilitando o "webgl.force-enabled" digitando "about:config" na barra de endereço no navegador [10].

No Safari a WebGL é compatível com Mac OS X 10.6 e a partir do Safari 5.x Poderá ser habilitado através do menu Desenvolvedor [10].

No Chrome/Chromium a WebGL está disponível nas versões estáveis para *desktop*. A WebGL no Chrome funciona no Android que possui a extensão GLEXT. Porém o Chrome para iOS não, devido à restrição da plataforma. Há uma extensão chamada WebGL Inspector [27] para depurar, diagnosticar e explorar cenas WebGL [10].

Já no Opera, a WebGL é suportada a partir da versão 12.0. Para habilitar aceleração de hardware e a WebGL deve digitar "opera:config" na barra de endereço no navegador e ativar. Após isso deve-se reiniciar o navegador [10].

3.5 Componentes de uma aplicação WebGL

Segundo Cantor [4] (2012) a estrutura de uma aplicação WebGL possui os seguintes componentes:

- Canvas: É o espaço reservado onde a cena será processada. Ele é um padrão Elemento HTML5 e como tal, ele pode ser acessado usando o Document Object Model (DOM), através de JavaScript.
- Objetos: Estes são as entidades 3D que fazem parte da cena.
- Luzes: WebGL usa *Shaders* para modelar luzes na cena.

[3] É uma extensão Mesa que permite que os programas renderizem para um *buffer off-screen* usando a API OpenGL sem ter que criar um contexto de renderização em um servidor X [31]

- Câmera: O canvas funciona como a porta de visualização para o mundo 3D. Diferentes operações de matriz são necessárias para produzir uma perspectiva de visão.

Parisi [17] (2014) descreve a anatomia de uma aplicação WebGL citando oito passos mínimos que a aplicação deverá conter para renderizar WebGL em uma página:

1. Criar um elemento canvas
2. Obter um contexto de desenho no canvas
3. Inicializar a *Viewport*
4. Criar um ou mais *buffers* contendo os dados a serem renderizados
5. Criar uma ou mais matrizes para definir a transformação dos *buffers* de vértice para espaço na tela.
6. Criar um ou mais *Shaders* para implementar o algoritmo de desenho
7. Inicializar os *Shaders* com parâmetros
8. Desenhar

3.6 Depuração do código

O WebGL Inspector é uma ferramenta com o objetivo de tornar o desenvolvimento de aplicações avançadas WebGL mais fáceis. Ele fornece um conjunto interativo de ferramentas para depuração e diagnóstico de aplicações avançadas WebGL. O WebGL Inspector possui a capacidade de capturar chamadas de renderização de imagens inteiras e de forma interativa passar por elas. Também possui um registo de chamadas com navegação passo-a-passo/recursos e avisos de chamadas redundantes, navegadores de recursos para texturas, *buffers* e programas, e muito mais. O WebGL Inspector está disponível para os principais navegadores [27].

O mecanismo de relatório de erros da WebGL envolve chamar a função *getError* para verificação de erros. Como pode ser onerosa colocar uma chamada *getError* depois de cada função chamada na WebGL, existe uma pequena biblioteca para tornar isso mais fácil: a biblioteca webgl-debug.js distribuída pelo Khronos Group. Para usar a biblioteca, basta baixá-la, colocá-la em seu servidor e incluí-la como mostra a Listagem 3.1.

```
1 <script src="webgl-debug.js"> </script>
```

Listagem 3.1: Usando a WebGL-Debug.

3.7 Uso da WebGL

A WebGL pode ser utilizada para diversos tipos de aplicações e em diversas áreas como educação, comércio, artes e cinema. Alguns usos da WebGL podem ser vistos nas Figuras 3.1 - 3.4.

Figura 3.1: Passeios virtuais: Como o da Catedral de Saint Jean em Lyon [5].

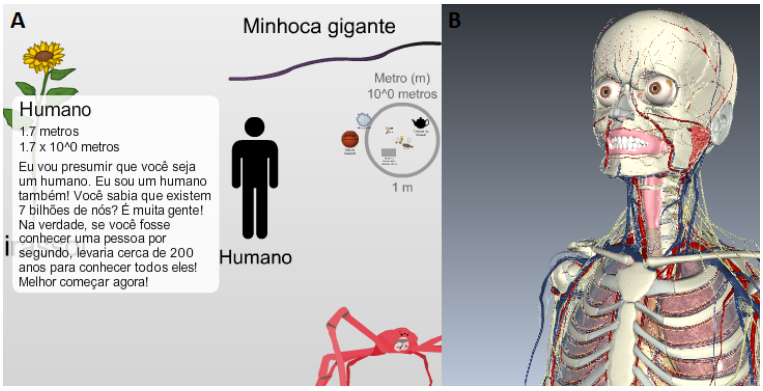

Figura 3.2: Educação: A Escala do Universo [13] e o Corpo Humano em 3D [2] são exemplos.

Figura 3.3: Jogos: Um dos jogos mais famosos também na versão WebGL [8].

Figura 3.4: Arte: Schwartz et al. [19] (2013) escreve sobre museus e patrimônio cultural e a necessidade de apresentações interativas e foto-realísticas das réplicas de artefatos em 3D. Desde artes plásticas até artefatos arqueológicos. O desafio não é apenas representar em 3D mas sim reproduzir fielmente todo sua aparência real.

Ainda citando exemplos do uso da WebGL pode destacar a visualização de dados, como exames de tomografia e ressonância magnética: Congote et al. [6] (2011) citam a dificuldade de armazenar, distribuir e de ter uma visualização interativa dessas imagens em 3D, que requerem computadores especializados, pois existem poucas soluções para computadores comuns. Isso também acontece com imagens geradas por radares. Eles enfatizam que a WebGL é o nome padrão para aceleração de gráficos 3D e também uma solução possível para essas dificuldades.

Outros usos da WebGL são: na criação de vídeos, onde pode facilitar a captura de movimentos para a produção de filmes [14]; na arquitetura para visualização de modelos arquitetônicos [25] e no comércio de veículos, como pode ser visto no site [18] que ilustra vários modelos de carros em 3 dimensões.

3.8 *Pipeline* da WebGL

Antes de começar a falar de *shaders* e do *pipeline* da WebGL é necessário voltar ao OpenGL e explicar as mudanças que ocorreram com o tempo.

Segundo Shreiner (et al. 2013) [23] o sistema de gráficos OpenGL é : "uma interface de software para o hardware gráfico. O GL representa *Graphics Library* (Biblioteca Gráfica). Ele permite que você crie programas interativos que produzem imagens coloridas em movimento de objetos tridimensionais. Com OpenGL, você pode controlar a tecnologia em computação gráfica para produzir imagens realistas ou aqueles que partem da realidade de forma imaginativa."

No começo do OpenGL a sua *pipeline*[4] era fixa. O usuário apenas podia usá-la mas não editá-la. A Figura 3.5 mostra como era o OpenGL na versão 1.1:

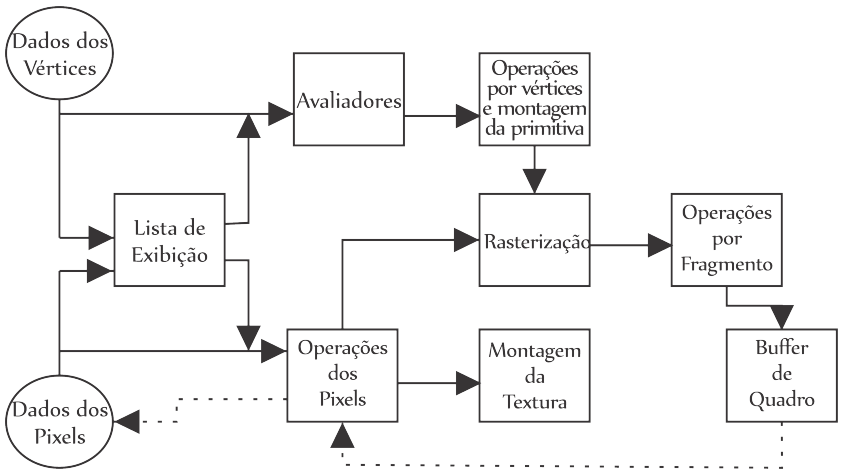

Figura 3.5: *Pipeline* do OpenGL na versão 1.1.

O diagrama da Figura 3.5 mostra que OpenGL leva para processamento de dados. Dados geométricos (vértices, linhas e polígonos) seguem o caminho através da linha de estágios que inclui avaliadores e operações por vértices, enquanto os dados de *pixel* (*pixels* e imagens) são tratados de forma diferente por uma parte do processo. Ambos os tipos de dados passam pelas mesmas

[4]É um método de execução de lista de processos onde eles são divididos em subprocessos para otimizar a execução [20].

etapas finais (rasterização e operações por fragmento) antes dos *pixels* finais ser escrito no *buffer* de quadro [22].

As últimas atualizações transformaram o *pipeline* fixo em *pipeline* programável, fazendo o programador do OpenGL utilizar *shaders* para processar os vértices, fragmentos, geometria e avaliar a tesselação. A Figura 3.6 mostra como o *pipeline* na versão 4.3 do OpenGL.

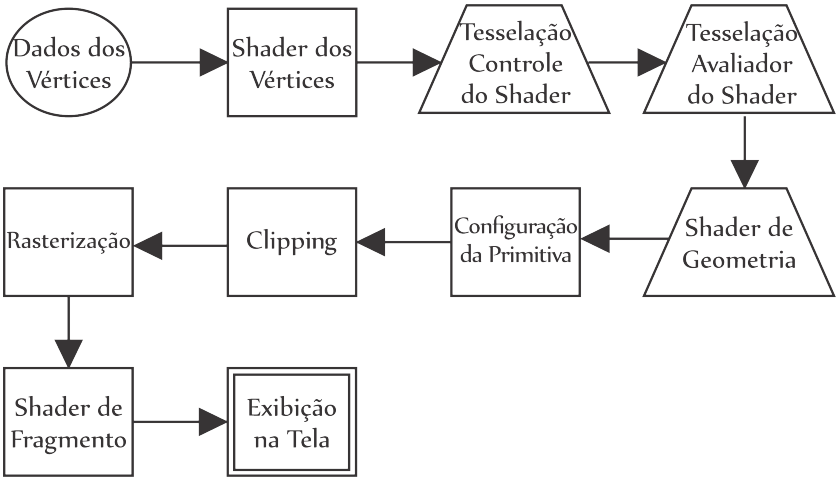

Figura 3.6: *Pipeline* do OpenGL na versão 4.3.

O diagrama da Figura 3.6 mostra que OpenGL começa com os dados geométricos (vértices e primitivas geométricas) e primeiro processa estes valores através de uma sequencia de estágio de *shaders*: *shaders* de vértices, tesselação e geometria. O rasterizador irá gerar fragmentos para toda primitiva dentro da região do *clipping*[5] e executará um *shader* de fragmento para cada fragmento gerado. Os *shaders* são essenciais para a criação de qualquer aplicação em OpenGL e o programador tem total controle sobre eles [23].

Como já foi dito anteriormente, a WebGL foi originalmente baseada em OpenGL ES 2.0, a versão de especificação OpenGL para sistemas embarcados. Mas como a especificação evoluiu, tornou-se independente e com o objetivo de oferecer a portabilidade em vários sistemas operacionais e dispositivos [4]. A renderização em WebGL requer o uso de *shaders* que estão em

[5]É uma operação que evita que os vértices da primitiva ultrapasse a *viewport*.

conformidade na versão 1.0 da linguagem de *shader* do OpenGL ES 2.0 - a GLSL [11].

3.9 Matrizes de Transformação

Na computação gráfica as transformações usadas são: escala, translação, rotação, reflexo e cisalhamento de forma e objetos. Essas transformações alteram os valores das coordenadas dos objetos [30]. Para essas operações podem ser utilizadas as matrizes, pois são mais fáceis de usar e entender do que equações algébricas. As transformações dos objetos são feitas pela multiplicação da matriz pelo seu vetor de vértices. As matrizes pode combinar várias transformações simultâneas. [1].

Segundo (Azevedo e Conci, 2003) [1] (na versão antiga do OpenGL) : "O OpenGL mantém três matrizes de transformação *ModelView*, *Projection* e *View-Point*, que são usadas para transformar um ponto qualquer dado em um ponto da janela de visualização. Cada ponto especificado é multiplicado por essas três matrizes". Na WebGL é diferente, o usuário deve criar matrizes e enviar até a GPU para transformar os vértices. Isso é feito através dos *shaders* que serão descritos na seção 3.10.

Neste projeto utilizou-se a biblioteca JavaScript glMatrix[6] que permite cálculos de vetores e matrizes de uma forma mais simples e rápida.

3.10 *Shaders*

Os *shaders* são escritos em GLSL, a linguagem de *Shading* do OpenGL. Ela é uma linguagem especialmente projetada para gráficos e é similar a linguagem C com um pouco de C++ misturado [23]. A Figura 3.6 mostra o *pipeline* de renderização, nele há quatro estágios no qual o usuário da API controla providenciando os *shaders* [23]:

- O estágio de *shader* dos vértices que é especificado através de VBO[7]. Os

[6] glMatrix - http://glmatrix.net/
[7] *Vertex Buffer Object*

vértices nas coordenadas do objetos são transformados nas coordenadas da tela.

- O estágio de *shader* de tesselação (opcional) que gera geometrias adicionais dentro do *pipeline* do OpenGL.
- O estágio de *shader* de geometria (opcional) que pode modificar primitivas geométricas inteiras dentro do *pipeline* do OpenGL.
- E por fim o estágio de *shader* de fragmento processa os fragmentos individuais gerados pelo rasterizador. É onde as cores e a profundidade dos fragmentos são computados.

A forma que o usuário da WebGL passa os valores para os *shaders* é através do VAO[8]. Os *buffers*, informações de estado e dos programas dos *shaders* são guardadas nele. Cada *buffer* de dados dos vértices é um *Vertex Buffer Object* - VBO [15].

3.10.1 *Vertex Buffer Objects - VBO*

Cada VBO armazena dados de um atributo particular dos vértices (posição, cor, coordenadas de texturas, etc). Para criar um VBO usa-se a função: *gl.createBuffer();*.

Após a criação utiliza-se a função *gl.bindBuffer(Parametro1, Parametro2);* para associar que possui dois parâmetros, onde o primeiro por ser:

- *gl.ELEMENT_ARRAY_BUFFER*: índices de vértices
- *gl.ARRAY_BUFFER*: atributos dos vértices

E o segundo é a instância do *buffer* criada anteriormente. Após, pode-se copiar dados para o *buffer* com a função *gl.bufferData(Parametro1, Parametro2, Parametro3)*. O primeiro parâmetro é igual o da função anterior *gl.bindBuffer*. O segundo são os dados e o terceiro pode ser:

[8] *Vertex Array Object*

- *gl.STATIC_DRAW*: os dados serão estáticos na aplicação e será utilizado muitas vezes.
- *gl.DINAMIC_DRAW*: toda vez que utilizar deverá definir os dados.
- *gl.STREAM_DRAW*: os dados serão estáticos na aplicação e será utilizado poucas vezes.

A Listagem 3.2 mostra como é a criação do VBO.

```
var dados = [ 0.0, 1.0, 0.0,
1.0, 1.0, 1.0,
0.0, 1.0, 0.0
];
var meuBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, meuBuffer);
gl.bufferData(gl.ARRAY_BUFFER, dados, STATIC_DRAW);
```

Listagem 3.2: Script de criação do VBO.

3.10.2 Tipos de primitivas

Na WebGL existem três tipo de primitivas: pontos, linhas e triângulos, como mostra a Figura 3.7.

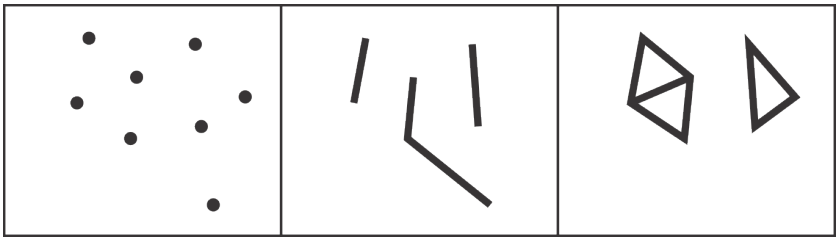

Figura 3.7: Primitivas da WebGL.

A partir das variações das primitivas qualquer coisa pode ser desenhada. As variações das linhas são: LINE___STRIP e LINE___LOOP. LINE___STRIP uma coleção de vértices em que, exceto para a primeira linha, o vértice de inicio da próxima linha é o final da linha anterior. Já LINE___LOOP é

o mesmo que o LINE___STRIP porém o ciclo é fechado quando o último vértice se conecta ao primeiro. A Figura 3.8 ilustra essas variações:

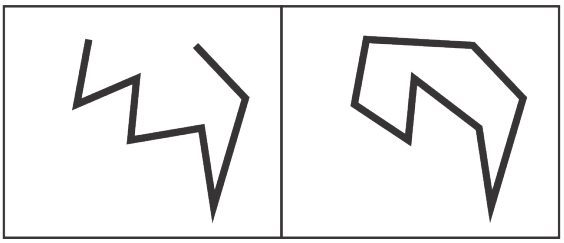

Figura 3.8: Variações da linha.

Uma variação do triângulo é TRIANGLE___STRIP utiliza os últimos dois vértices, juntamente com o vértice seguinte para formar triângulos. A outra é o TRIANGLE___FAN que utiliza o primeiro vértice para todos os triângulos. A Figura 3.9 ilustra essas variações:

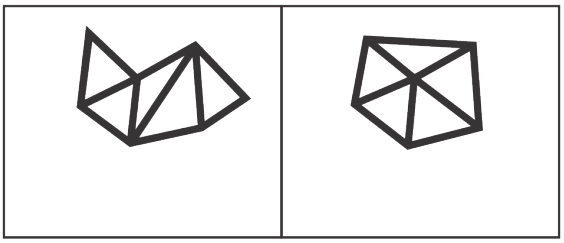

Figura 3.9: Variações do triângulo.

4 Visualizador de Modelos WebGL

4.1 Introdução

A ideia deste trabalho é implementar um Visualizador de Malhas tridimensionais para exemplificar o uso da WebGL, suas capacidades e seus potenciais. Um visualizador é uma aplicação para renderizar malhas triangulares não estruturadas. Essas malhas podem ter sido modeladas em *softwares* de modelagem 3D como O Blender[1]. O visualizador carrega modelos do usuário e armazena-os no servidor. Com os modelos carregados o usuário pode alternar entre os *shaders*: Flat, Phong e sem tonalização. Um *software* muito conhecido de manipulação de malhas é o MeshLab[2]. Neste trabalho será implementado um visualizador de malhas com a vantagem de ser uma aplicação *web*, pois não necessita de instalação e pode ser acessado de qualquer dispositivo que tem acesso à internet e suporte à WebGL.

4.2 Representação do Modelo

Nesta implementação utilizou-se a estrutura de dados *Halfedge* para armazenar o modelo na memória. As estruturas do tipo *edge-based* colocam as informações de conectividade nas arestas. A Figura 4.1 ilustra a forma como a conectividade é armazenada nesta estrutura.

[1]https://www.blender.org/
[2]http://meshlab.sourceforge.net/

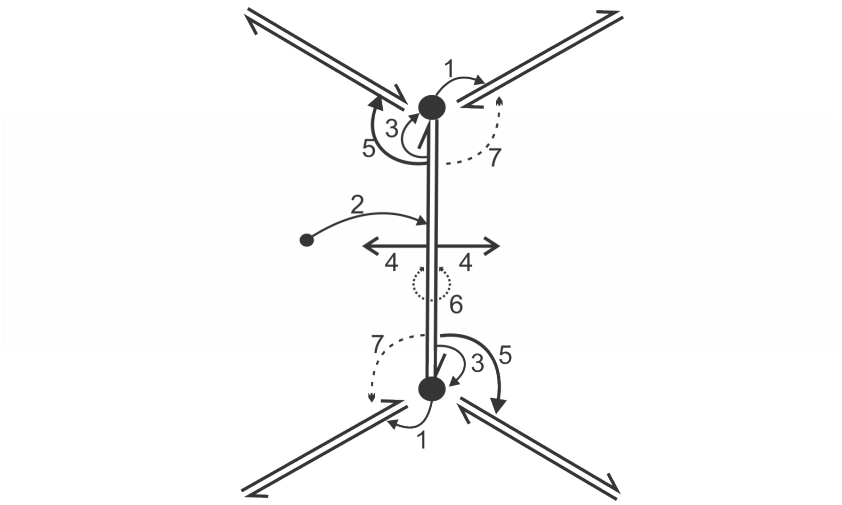

Figura 4.1: A estrutura de dados *Halfedge*[3].

Cada aresta referencia seus dois vértices, as faces às quais pertence, uma próxima aresta na face e opcionalmente uma aresta com sentido oposto. Assim, conforme a Figura 4.1:

1. Cada vértice referencia uma *halfedge*.
2. Cada face referencia a uma das *halfedges* que a delimita.
3. Cada *halfedge* fornece um identificador para o vértice que aponta.
4. Cada *halfedge* fornece um identificador para a face a qual pertence.
5. Cada *halfedge* fornece um identificador para a próxima *halfedge* da face.
6. Cada *halfedge* fornece um identificador para a *halfedge* oposta.
7. Cada *halfedge* fornece um identificador para a *halfedge* anterior na face (opcional).

As estruturas do tipo *edge-based* geralmente consomem mais memória em relação às outras estruturas, mas em contrapartida elas possuem algumas vantagens:

- É fácil acessar as faces que compartilham um vértice;

- Com a explícita representação de vértices, faces e arestas (*halfedge*) são extremamente úteis para armazenar dados por aresta (*halfedge*);
- Pode-se circular um vértice para obter as faces às quais este pertence facilmente.

Na implementação deste projeto, são armazenados os vértices, as faces e também as arestas. Para facilitar na etapa de iluminação, cada face e vértice já possuem suas normais associadas. As arestas possuem como atributos o vértice de origem e o vértice destino, armazenando apenas seus índices e também possuem uma aresta oposta. Isto permite circular faces próximas.

Todo vértice possui seu ponto, uma lista dos índices das faces que pertence (auxilia no cálculo de sua normal) e uma aresta de saída. A aresta associada ao vértice pode permitir encontrar vértices adjacentes e também as faces das quais o vértice faz parte.

As faces necessitam apenas o índice de uma de suas arestas. Também é armazenada a normal na face, para facilitar e agilizar os cálculos da iluminação.

Além das vantagens acima citadas a ideia de utilização desse modelo de estrutura era permitir filtros. Outra vantagem do *Halfedge* é permitir iterar por faces, vértices e obter arestas e vértices próximos a um vértice.

4.3 Renderização

Com o modelo carregado na estrutura de dados, a aplicação calcula as normais das faces e dos vértices. Esses cálculos são fundamentais para a modelagem do objeto na cena, pois influenciam diretamente na iluminação e tonalização. Para renderizar o objeto na *viewport* é selecionado o *shader* e são passados os parâmetros a ele.

4.3.1 Iluminação e Tonalização

Iluminação em computação gráfica é fundamental para que os objetos 3D tenham dimensão e profundidade na cena. Para demonstrar essa afirmação

foi utilizado o exemplo de iluminação do Danchilla (2012) [7]. A Figura 4.2 mostra um desenho sem iluminação:

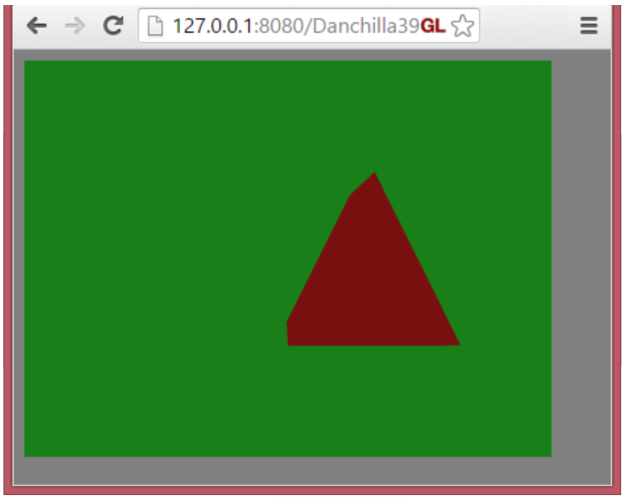

Figura 4.2: Um desenho sem iluminação [7].

O que é observado é um desenho 2D simples com um fundo verde. A Figura 4.3 mostra o mesmo desenho agora com iluminação:

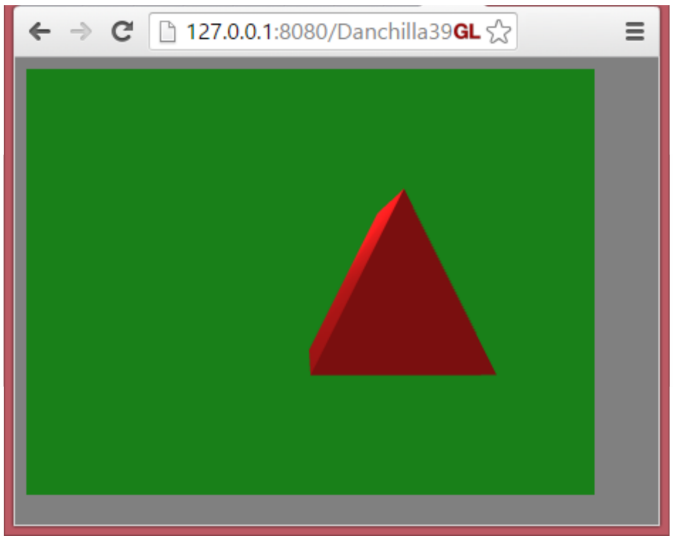

Figura 4.3: Um desenho com iluminação [7].

Na Figura 4.3 fica mais fácil de ver o prisma triangular devido à iluminação.

De uma forma geral vamos destacar três tipos de iluminação para poder introduzir o modelo de iluminação utilizado neste projeto. O primeiro tipo, a iluminação ambiente, atinge todo o objeto igualmente em todas as direções [1]. O resultado é mostrado na Figura 4.4:

Figura 4.4: Iluminação ambiente [26].

O segundo tipo é a iluminação difusa onde cada superfície do objeto serve como difusor da luz e determina quanto da luz será refletida. Segundo Azevedo e Conci (2003) [1]: "A quantidade de luz refletida não depende do ângulo de visão da superfície, mas sim de sua orientação em relação a direção de luz." . O resultado do segundo tipo é mostrado na Figura 4.5:

Figura 4.5: Iluminação difusa [26].

Por fim, o terceiro tipo é a iluminação especular, segundo Azevedo e Conci (2003) [1]: "Nesse tipo de reflexão, o fóton não interage com os pigmentos da superfície deixando a cor da luz refletida igual à cor original da luz incidente. Basicamente, determinados pontos da superfície atuam como espelho refletindo a luz incidente sem atenuações". A iluminação especular é mostrada na Figura 4.6:

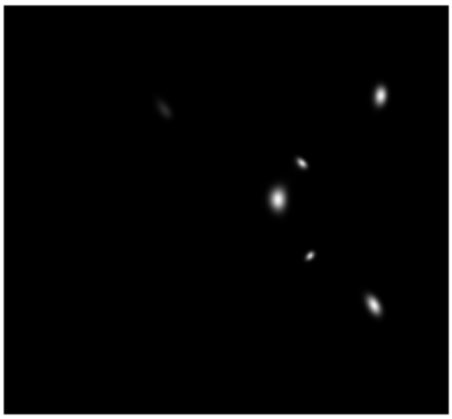

Figura 4.6: Iluminação especular [26].

O modelo de iluminação utilizado nesse projeto é o Phong [1] (Figura 4.7) ele utiliza os três tipos de iluminação citados anteriormente. Deve-se gerar as normais da face e dos vértices do objeto antes de aplicar um modelo de iluminação.

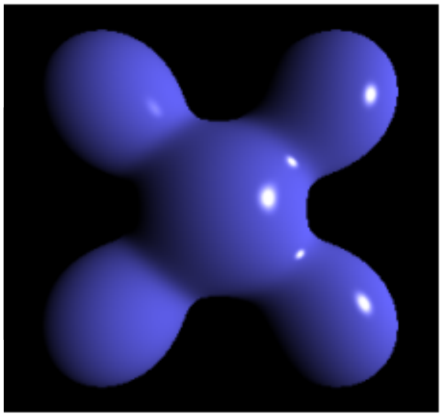

Figura 4.7: Modelo de iluminação Phong [26].

Normal

Geralmente, na iluminação, um vértice terá um vetor normal associado a ele, de modo que no cálculo de iluminação seja possível saber em qual direção a superfície reflete a luz [23]. A normal de um triângulo é um vetor de tamanho 1 (um) que é perpendicular ao triângulo. Ele é calculado tomando o produto cruzado de duas de suas arestas.

O modelo de iluminação utilizado nesse projeto (Phong) usa as normais dos vértices para fazer o calculo da iluminação. Para calcular a normal do vértice é necessário a combinação das normais das faces dos triângulos que compartilham o vértice. A Figura 4.8 exemplifica isto:

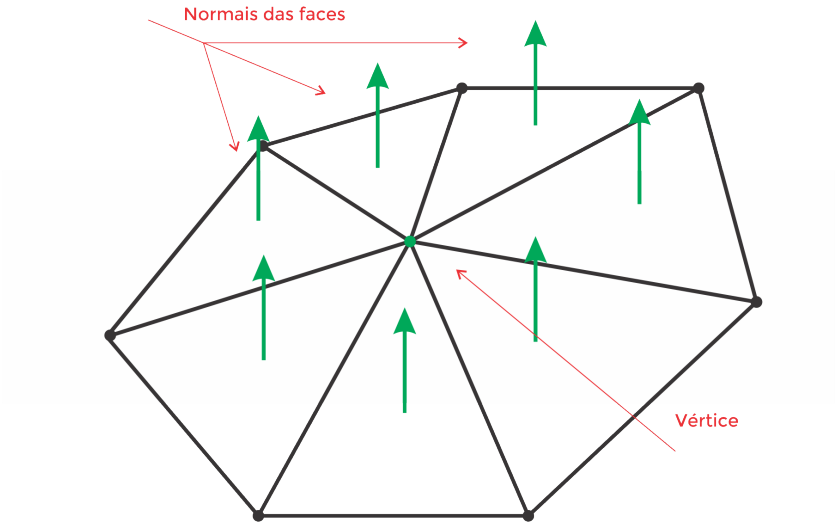

Figura 4.8: O vértice e suas faces.

Neste projeto a normal de cada vértice é calculada no momento do carregamento da malha quando é gerado o *halfedge*. A Listagem 4.1 mostra como é calculada a normal da face:

```
1 var e1v1= this.vertices[edge1.originVertex];
2 var e1v2= this.vertices[edge1.targetVertex];
3 var vec1 = new Vector(e1v2.x-e1v1.x, e1v2.y-e1v1.y,e1v2.z-e1v1.z
    );
4
5 e1v1= this.vertices[edge3.targetVertex];
6 e1v2= this.vertices[edge3.originVertex];
```

```
7  var vec2 = new Vector(e1v2.x-e1v1.x, e1v2.y-e1v1.y,e1v2.z-e1v1.z
       );
8
9  var faceN = vec1.cross(vec2);
10 faceN.normalize();
11 this.faces[i].normal = faceN;
```

Listagem 4.1: Calcula a normal da face.

A normal da face é o produto cruzado dos vetores formados pelos vértices da face. Após calculada a normal de cada face são calculadas as normais dos vértices. A Listagem 4.2 possui dois laços *for* que mostram como são calculadas as normais de cada vértice:

```
1  for(i = 0; i < this.vertexNumber; i++)
2      {
3          var j;
4          var v = new Vector()
5          var numFaces = this.vertices[i].faces.length;
6
7          for(j = 0; j < numFaces; j++)
8          {
9              v = v.add(this.faces[this.vertices[i].faces[j]].
       normal);
10         }
11
12         v.normalize();
13         this.vertices[i].normal = v;
14     }
```

Listagem 4.2: Calcula as normais de cada vértice.

O primeiro laço percorre todos os vértices e o segundo adiciona todas as face do vértice a ser calculado. A normal do vértice é a média das normais das faces.

4.3.2 *Shaders*

Neste projeto foram implementados três *shaders*. Um deles apenas repassa a cor do vértice para o fragmento; assim não é possível identificar as

formas do modelo, apenas seu contorno. O resultado é o *Pass-thru* semelhante ao da Figura 4.4.

Outro *shader*, o *Flat* mostra o modelo facetado, assim é fácil visualizar os seus triângulos. Para essa implementação é necessário que seja utilizada a normal da face para todos os vértices, assim a iluminação fica constante dentro de cada triângulo [7]. Este triângulo ficará com uma cor sólida, deixando-o bem evidente como mostra a Figura 4.9:

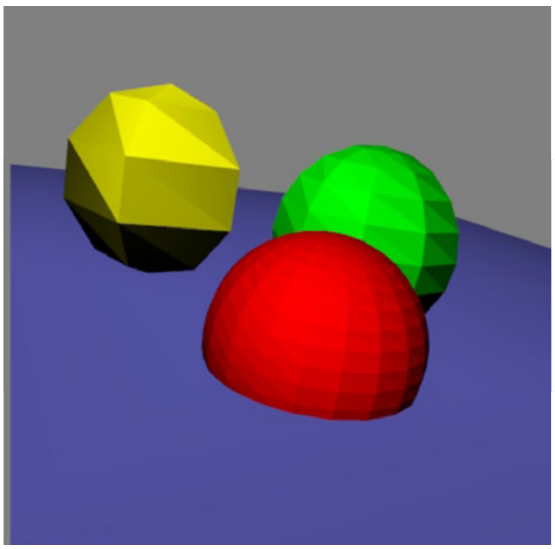

Figura 4.9: O *shader* Flat [7].

O Modelo Phong utiliza as normais por vértice, assim, a iluminação dentro de cada triângulo é suave como já foi mostrado na Figura 4.7.

4.4 Navegação

Para a manipulação de rotação do modelo 3D é utilizada a técnica *Virtual Trackball*. A *Virtual Trackball* (também conhecida como *Arcball*) funciona de maneira similar aos dispositivos *Trackball* físicos. Essa técnica permite que o usuário rotacione o objeto 3D usando o clique do mouse em uma tela 2D [9, 21]. A Figura 4.10 ilustra como a *Trackball* é representada na tela:

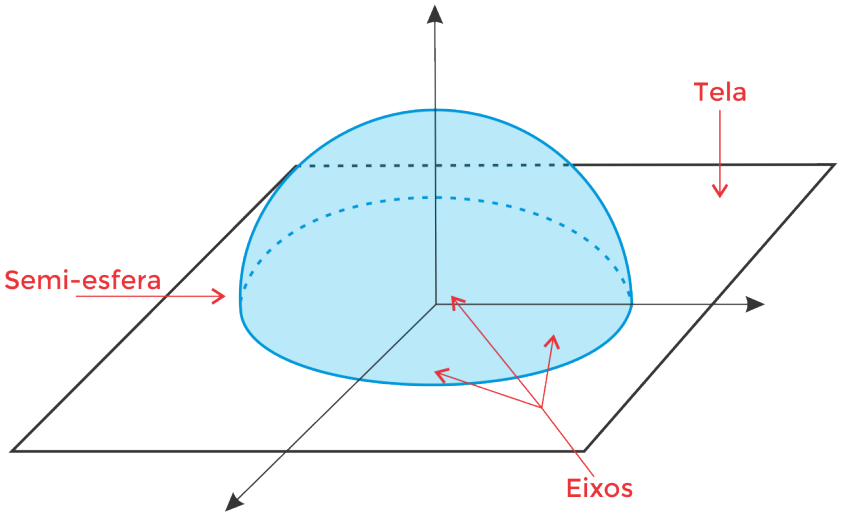

Figura 4.10: Representação do Trackball na tela.

Conforme a Figura 4.10 a *Trackball* projeta uma semiesfera em um circulo na *viewport*. A posição do mouse é projetada ortograficamente nesta semiesfera, dessa forma a *Virtual Trackball* consegue rastrear a posição anterior e a posição atual do mouse, fazer o calculo da projeção e após a rotação [9, 21].

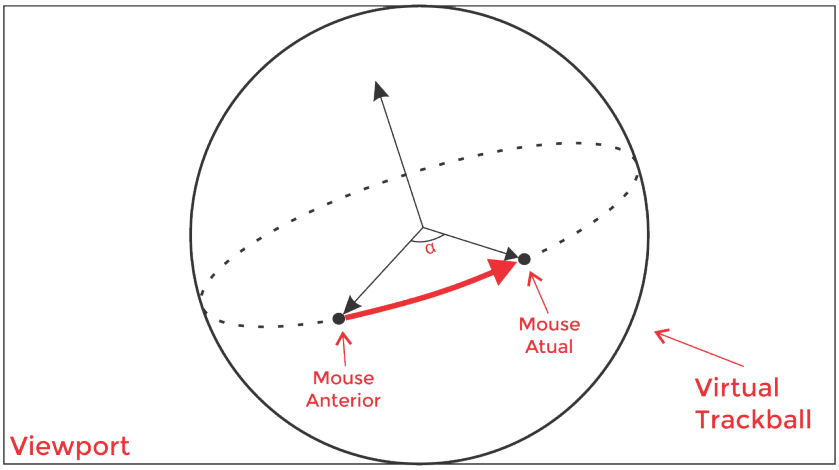

Figura 4.11: *Trackball* e a rotação do objeto 3D.

A Figura 4.11 exemplifica bem o que foi dito anteriormente. A *Trackball* pega o primeiro clique até o fim do arraste do mouse, calcula o angulo da projeção e faz a rotação [9].

5 Apresentação do Aplicação

5.1 Introdução

Para o desenvolvimento do aplicativo de visualização de malhas, além da API WebGL, foi empregado um modelo arquetípico de serviço *web* gratuito para Microsoft Windows que possui três componentes de código livre: o servidor HTTP Apache[1], o sistema de controle de banco de dados relacional MySQL[2] e a linguagem de programação PHP[3]. Esse modelo para Microsoft Windows chamado WAMP é baseado no modelo para Linux, o LAMP. A Figura 5.1 abaixo mostra o diagrama de relacionamento do banco de dados usado neste projeto:

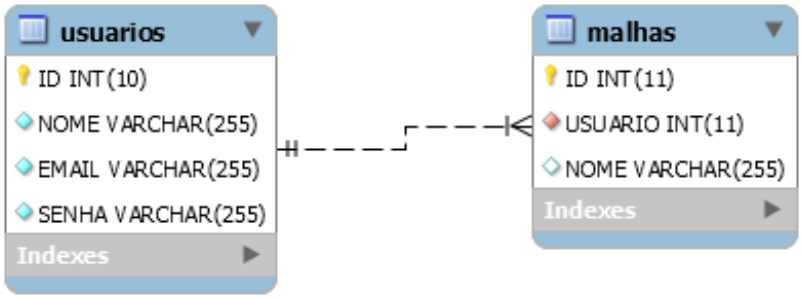

Figura 5.1: Diagrama de relacionamento.

[1]http://www.apache.org/
[2]https://www.mysql.com/
[3]http://php.net/

Para o *layout* da página foi utilizado o Bootstrap[4] que é um *framework front-end* de código livre. Também foi utilizado o jQuery[5], que é uma biblioteca JavaScript para alguns efeitos como o de transições para menu do aplicativo.

5.2 Telas

A Figura 5.2 mostra a tela inicial do aplicativo.

[4]http://getbootstrap.com/
[5]https://jquery.com/

Figura 5.2: Tela inicial do aplicativo.

Ao clicar no botão “Visualizador” ele deverá abrir uma tela conforme a Figura 5.3.

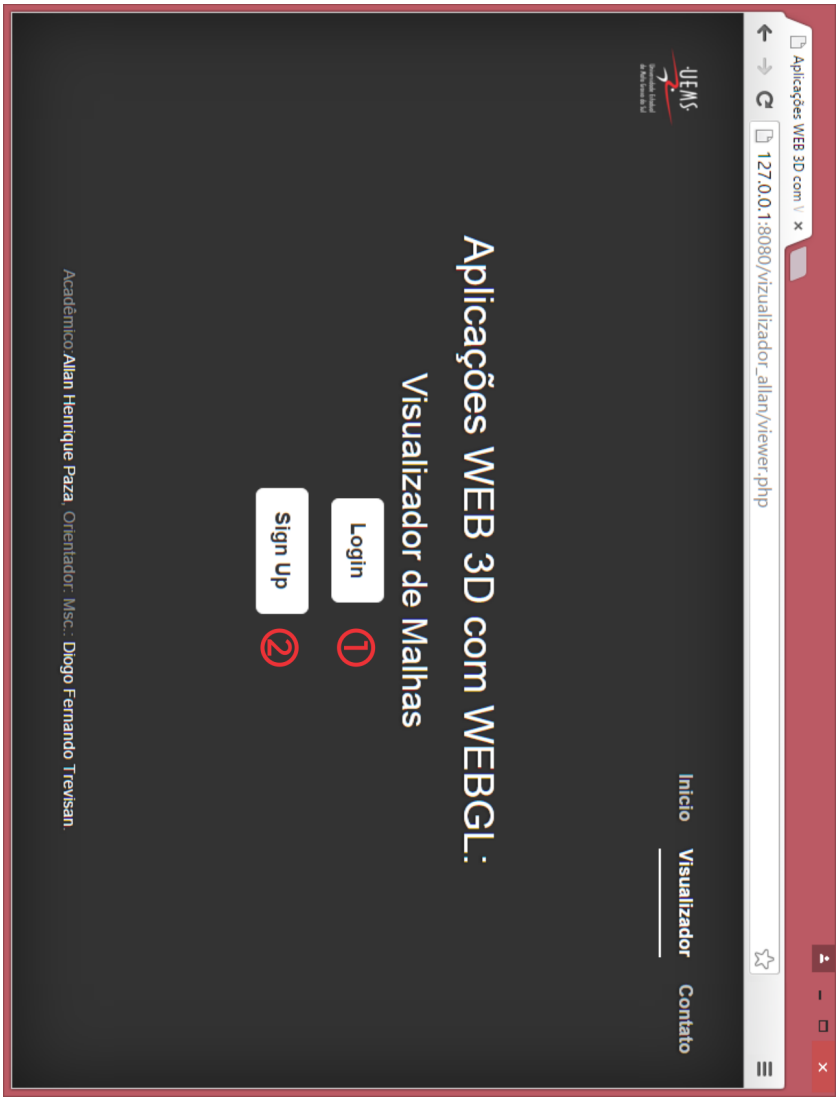

Figura 5.3: Tela para o usuário decidir entre *login* ou cadastro.

O usuário terá duas opções:

1. Fazer o login no sistema;
2. Criar um login.

Ao selecionar a primeira opção aparecerá um formulário conforme a Figura 5.4.

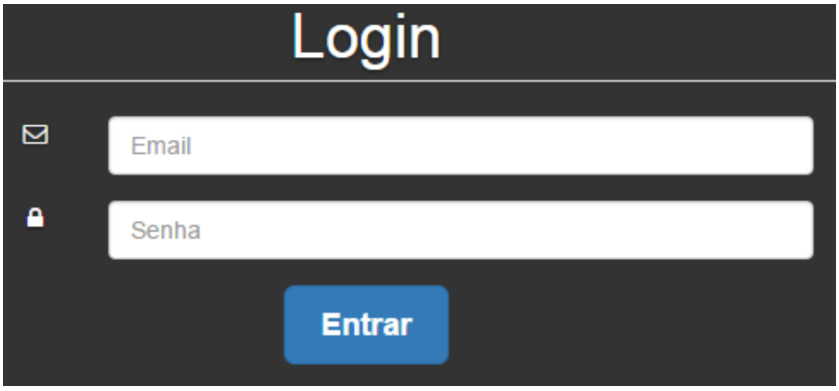

Figura 5.4: Formulário para *login* no aplicativo.

Caso o usuário selecione a segunda opção aparecerá um formulário conforme a Figura 5.5.

Figura 5.5: Formulário para cadastro no aplicativo.

Após o usuário finalizar o cadastro ele será redirecionado para a tela da Figura 5.3. Neste tala o usuário poderá efetuar o login (opção 1). Com o login efetuado a próxima tela é da Figura 5.6.

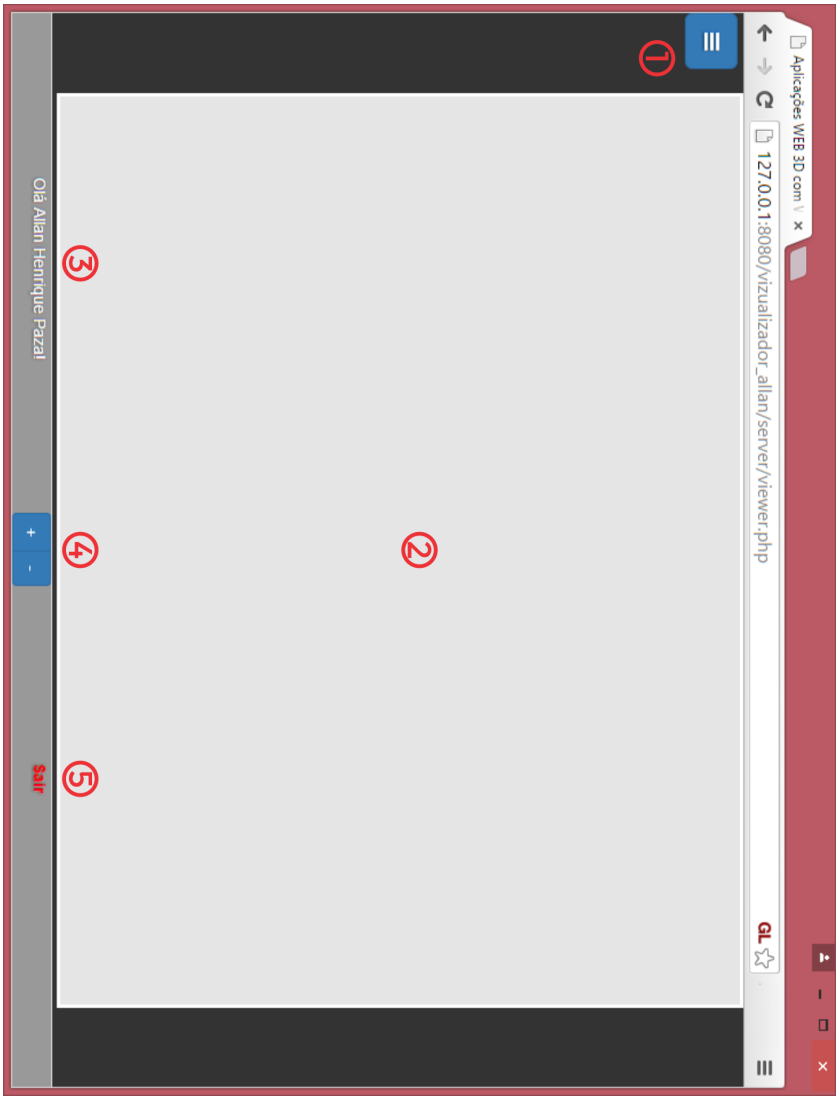

Figura 5.6: Tela do visualizador.

A tela do visualizador (Figura 5.6) possui os seguinte itens:

1. Botão para abrir o menu;
2. *Viewport*;
3. Mensagem de boas vindas;
4. Botões de zoom;
5. Botão de *logout*.

Quando o usuário clica no botão para abrir o menu, o seguinte menu é mostrado (Figura 5.7).

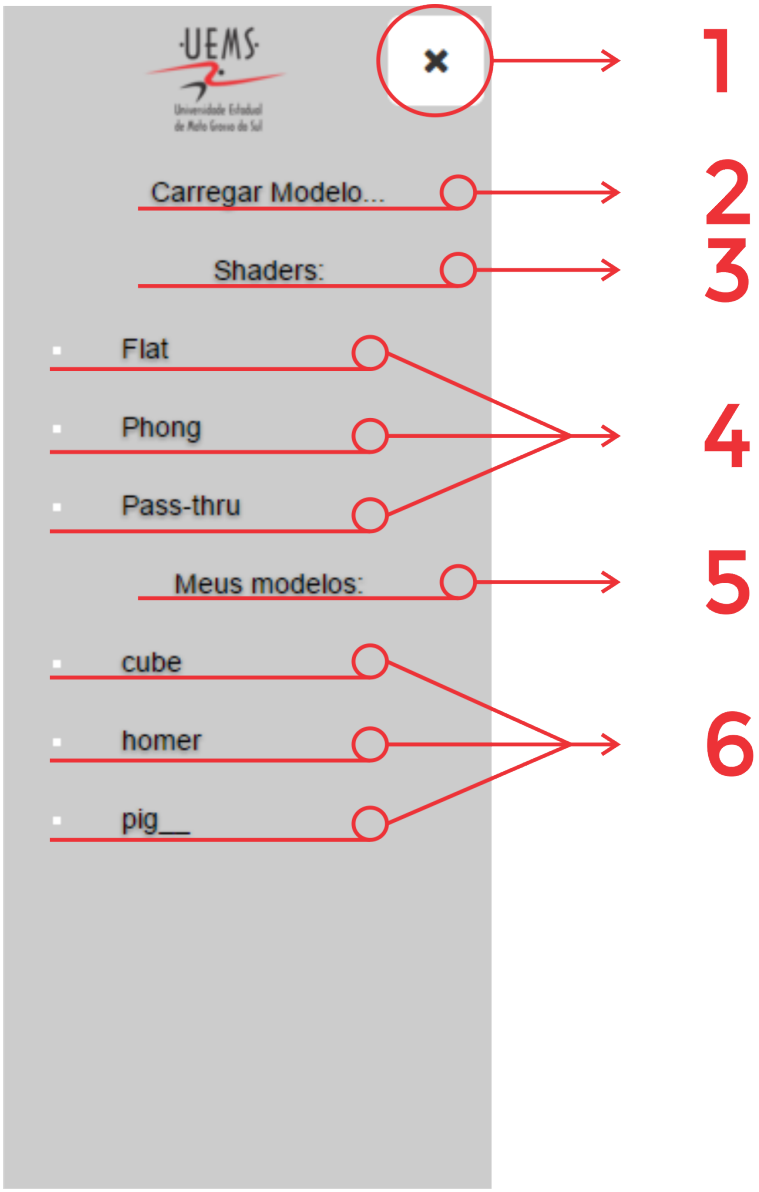

Figura 5.7: Menu do visualizador.

O menu do visualizador (Figura 5.7) possui os seguinte itens:

1. Botão para fechar o menu;
2. Botão para abrir a tela de *upload* de malhas;

3. Botão para listar os *shaders*;
4. *Shaders* disponíveis;
5. Botão para listar as malhas do usuário;
6. Lista de malhas do usuário.

Antes de mostrar a tela de *upload* de malhas (item 2) serão explicados os outros botões do menu. No item 3, o botão lista os *shaders* implementados neste projeto conforme dito no capítulo 4.3.2. A Figura 5.8 mostra um objeto renderizado utilizando os três *shaders*: Flat, Phong e Pass-thru respectivamente.

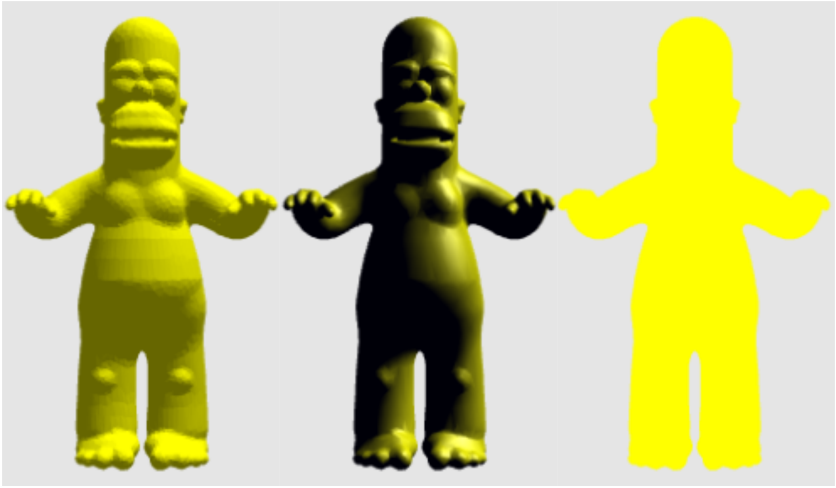

Figura 5.8: Flat, Phong e Pass-thru respectivamente.

No item 5, o botão abre a lista de todas as malhas que o usuário carregou no aplicativo. Caso ele não tenha carregado nenhuma será mostrado uma mensagem de que não ele não possui malhas.

E por fim, no item 2 deverá abrir a tela da Figura 5.9.

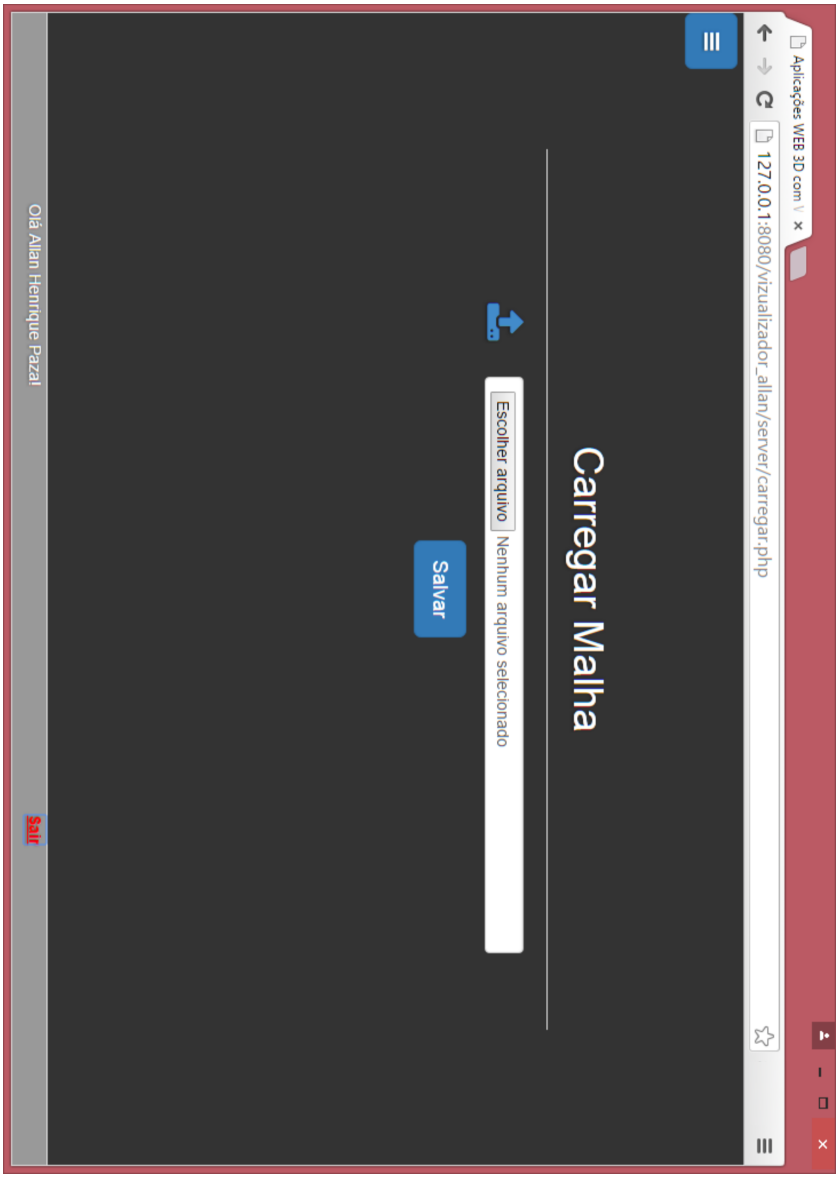

Figura 5.9: Tela de carregamento de malhas.

O usuário escolhe a malha a ser carregada e clica no botão “Salvar”. O visualizador implementado nesse aplicativo suporta apenas arquivos com extensão “.OFF”.

6 Conclusões

A API WebGL mostrou-se uma ferramenta de grande potencial, pois o visualizador de malhas proposto e implementado neste trabalho pode ser considerado tão rápido e capaz quanto um visualizador *stand-alone*. Foi observado que desde a proposta deste trabalho até o seu final, uma versão da WebGL (1.0.3) foi finalizada e uma nova esta em desenvolvimento (1.0.4), isto mostra o desenvolvimento ativo, um fator de alta importância para qualquer ferramenta de programação.

Com o crescimento tanto do uso e do poder de processamento dos dispositivos móveis, da melhoria da *internet* móvel e do desenvolvimento ativo a WebGL tem grandes chances de ser um poderoso recurso para aplicações *web* e deve auxiliar em todas as áreas: da educação ao comércio.

Contudo, uma aplicação *web* usando a WebGL requer esforço e tempo consideráveis, ou seja, não pode-se criar aplicação 3D para *internet* rapidamente. Conforme foi dito em capítulos anteriores, além de o usuário estudar o tipo de aplicação que ele irá implementar, ele deve entender GLSL e operações de matrizes.

Para trabalhos futuros, o visualizador de malhas poderá aplicar texturas e também utilizar outros *shaders*.

Bibliografia

[1] E. Azevedo and A. Conci. *Computação gráfica: teoria e prática*. Elsevier, Rio de Janeiro - RJ, 2003. ISBN 9788535212525.

[2] BioDigital. *BioDigital Human: Explore the Body in 3D!*, 2014 (acessed July 11, 2014). URL https://human.biodigital.com/index.html. Tradução nossa.

[3] Mario Botsch, Stephan Steinberg, Stephan Bischoff, and Leif Kobbelt. Openmesh-a generic and efficient polygon mesh data structure. Citeseer, 2002.

[4] D. Cantor and B. Jones. *WebGL Beginner's Guide*. Learn by doing : less theory, more results. Packt Publishing, 2012. ISBN 9781849691734.

[5] Gaël Chaize. *Saint Jean Cathedral in WebGL*, 2014 (acessed July 11, 2014). URL http://patapom.com/topics/WebGL/cathedral/index.html. Tradução nossa.

[6] John Congote, Alvaro Segura, Luis Kabongo, Aitor Moreno, Jorge Posada, and Oscar E. Ruiz. Interactive visualization of volumetric data with webgl in real-time. In *3D Technologies for the World Wide Web, Proceedings of the 16th International Conference on Web 3D Techno-*

logy, Web3D 2011, Paris, France, June 20-22, 2011, pages 137–146, 2011. doi: 10.1145/2010425.2010449.

[7] Brian Danchilla. *Beginning WebGL for HTML5*. Apress, Berkely, CA, USA, 1st edition, 2012. ISBN 1430239964, 9781430239963.

[8] Rovio Entertainment. *Angry Birds Chrome*, 2014 (acessed July 11, 2014). URL http://chrome.angrybirds.com/. Tradução nossa.

[9] J.P. Gois and H.C. Batagelo. Interactive graphics applications with opengl shading language and qt. In *Graphics, Patterns and Images Tutorials (SIBGRAPI-T), 2012 25th SIBGRAPI Conference on*, pages 1–20, 2012. doi: 10.1109/SIBGRAPI-T.2012.10.

[10] Khronos Group. Getting a webgl implementation, 2014 (acessed July 11, 2014). URL http://www.khronos.org/webgl/wiki/Getting_a_WebGL_Implementation. Tradução nossa.

[11] Khronos Group. *WebGL - OpenGL ES 2.0 for the Web*, 2014 (acessed July 11, 2014). URL https://www.khronos.org/webgl/. Tradução nossa.

[12] Khronos Group. Getting started, 2014 (acessed July 11, 2014). URL http://www.khronos.org/webgl/wiki/Getting_Started. Tradução nossa.

[13] Cary Huang. *The Scale of the Universe*, 2014 (acessed July 11, 2014). URL http://htwins.net/scale2/lang.html. Tradução nossa.

[14] Catherine Leung and Andor Salga. Enabling webgl. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 1369–1370, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-799-8. doi: 10.1145/1772690.1772933.

[15] OpenGL.org. *Tutorial2: VAOs, VBOs, Vertex and Fragment Shaders (C / SDL) - OpenGL.org*, 2015 (acessed September 10, 2015). URL https://www.opengl.org/wiki/Tutorial2:_VAOs,_VBOs,_Vertex_and_Fragment_Shaders_(C_/_SDL). Tradução nossa.

[16] T. Parisi. *WebGL: Up and Running*. Oreilly and Associate Series. O'Reilly Media, Incorporated, 2012. ISBN 9781449323578.

[17] T. Parisi. *Programming 3D Applications with HTML5 and WebGL: 3D Animation and Visualization for Web Pages*. O'Reilly Media, 2014. ISBN 9781449363956.

[18] Plus360degrees. *360 Car Visualizer - Three.js*, 2014 (acessed July 11, 2014). URL http://carvisualizer.plus360degrees.com/threejs/. Tradução nossa.

[19] Christopher Schwartz, Roland Ruiters, Michael Weinmann, and Reinhard Klein. Webgl-based streaming and presentation of objects with bidirectional texture functions. *Journal on Computing and Cultural Heritage (JOCCH)*, 6(3):11:1–11:21, July 2013. ISSN 1556-4673. doi: 10.1145/2499931.2499932.

[20] J. Shen and M.H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. Electrical and Computer Engineering. McGraw-Hill Companies,Incorporated, 2005. ISBN 9780070570641. URL https://books.google.com.br/books?id=Nibfj2aXwLYC.

[21] Ken Shoemake. Arcball: a user interface for specifying three-dimensional orientation using a mouse. In *Graphics Interface*, volume 92, pages 151–156, 1992.

[22] D. Shreiner, G. Sellers, J.M. Kessenich, and B.M. Licea-Kane. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.1*. OpenGL. Addison-Wesley Pub, 1997. ISBN 9780201461381.

[23] D. Shreiner, G. Sellers, J.M. Kessenich, and B.M. Licea-Kane. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. OpenGL. Pearson Education, 2013. ISBN 9780132748438.

[24] Mauricio Samy Silva. *HTML5 - A Linguagem que Revolucionou a WEB*. Novatec, São Paulo - SP, 2011.

[25] Sketchfab. *Sketchfab - The web platform for publishing your interactive 3D models*, 2014 (acessed July 11, 2014). URL https://sketchfab.com/. Tradução nossa.

[26] Brad Smith. Illustration of the phong reflection model, 2015 (acessed October 16, 2015). URL https://upload.wikimedia.org/wikipedia/commons/6/6b/Phong_components_version_4.png. Tradução nossa.

[27] Ben Vanik. *WebGL Inspector*, 2014 (acessed July 11, 2014). URL http://benvanik.github.io/WebGL-Inspector/. Tradução nossa.

[28] Ben Vanik. *WebGL Conformance Tests*, 2015 (acessed October 18, 2015). URL https://www.khronos.org/registry/webgl/sdk/tests/webgl-conformance-tests.html. Tradução nossa.

[29] Ben Vanik. *WebGL Specifications*, 2015 (acessed October 18, 2015). URL https://www.khronos.org/registry/webgl/specs/latest/. Tradução nossa.

[30] J. Vince. *Mathematics for Computer Graphics*. Springer London, 2006. ISBN 9781849960229.

[31] VMware. *The Mesa 3D Graphics Library*, 2014 (acessed July 11, 2014). URL http://www.mesa3d.org/osmesa.html. Tradução nossa.

[32] Tim Wright. *Learning JavaScript: A Hands-On Guide to the Fundamentals of Modern JavaScript*. Learning. Pearson Education, 2012. ISBN 9780133016277.