

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de *Software*

Aproximação experimental da complexidade assintótica do *vertex* e *fragment shader* para plataforma *Android*

Autor: Aline de Souza Campelo Lima
Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF
2014



Aline de Souza Campelo Lima

Aproximação experimental da complexidade assintótica do *vertex* e *fragment shader* para plataforma *Android*

Monografia submetida ao curso de graduação
em Engenharia de *Software* da Universidade
de Brasília, como requisito parcial para ob-
tenção do Título de Bacharel em Engenharia
de *Software*.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF

2014

Aline de Souza Campelo Lima

Aproximação experimental da complexidade assintótica do *vertex* e *fragment shader* para plataforma *Android*/ Aline de Souza Campelo Lima. – Brasília, DF, 2014-

74 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2014.

1. Computação Gráfica. 2. Complexidade Algorítmica. I. Prof. Dr. Edson Alves da Costa Júnior. II. Universidade de Brasília. III. Faculdade UnB Gama.
IV. Aproximação experimental da complexidade assintótica do *vertex* e *fragment shader* para plataforma *Android*

CDU 02:141:005.6

Aline de Souza Campelo Lima

Aproximação experimental da complexidade assintótica do *vertex* e *fragment shader* para plataforma *Android*

Monografia submetida ao curso de graduação em Engenharia de *Software* da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de *Software*.

Trabalho aprovado. Brasília, DF, 02 de dezembro de 2013:

Prof. Dr. Edson Alves da Costa Júnior
Orientador

Prof. Dr. Ricardo Pezzoul Jacobi
Convidado 1

Prof. Dra. Carla Silva Rocha Aguiar
Convidado 2

Brasília, DF
2014

Resumo

A utilização dos dispositivos móveis e da plataforma *Android* tem crescido e constata-se a importância dos efeitos visuais em jogos, bem como a sua limitação atual de desempenho dos processadores gráficos destas plataformas. Assim, a proposta do trabalho se baseia na implementação e aproximação experimental da complexidade assintótica dos *shaders* (programas responsáveis pelos efeitos visuais) para a plataforma *Android*. Suas complexidades algorítmicas serão analisadas, baseando-se nas métricas de número de instruções por segundo e tempo de renderização em função da variação do número de polígonos renderizados. Além disso, o método dos mínimos quadrados será utilizado para ajustar os valores obtidos a uma curva, permitindo determinar qual curva mais se aproxima da função original.

Palavras-chaves: *Android*, *shaders*, dispositivos móveis, computação gráfica, jogos, complexidade algorítmica.

Abstract

The usage of mobile devices and Android platform is emerging and is notable the importance of visual effects in games and the performance restriction of the graphical processors of these platforms. This way, the purpose of this academic work is based on the development and experimental approximation of asymptotic computational complexity of shaders (programs responsible for the visual effects) for Android platform. Their algorithm complexities will be analyzed, based on number of instructions per second and rendering time metrics, depending on the number of polygons rendered. Besides, the method of least squares will be used to adjust the values obtained from a curve, being able to estimate which curve has better approximation to the original function.

Key-words: Android, shaders, mobile devices, computer graphics, games, algorithm complexity.

Listas de ilustrações

Figura 1 – Processo de renderização da <i>OpenGL</i>	21
Figura 2 – Utilização dos <i>shaders</i>	23
Figura 3 – Ambiente de desenvolvimento <i>Eclipse</i>	25
Figura 4 – Comparação entre as técnicas de <i>shading</i>	28
Figura 5 – Exemplo de escolha dos tons	29
Figura 6 – <i>Environment Map</i>	29
Figura 7 – Comparação da Complexidade Algorítmica	30
Figura 8 – Ferramenta <i>Adreno Profiler</i> : analisador de <i>shaders</i>	36
Figura 9 – Ferramenta <i>Adreno Profiler</i> : visualização de métrica quadros por segundo	37
Figura 10 – Diagrama de Classe da Implementação em <i>Android</i>	38
Figura 11 – Detalhamento das classes <i>Shader Activity</i> , <i>Splash Activity</i> e <i>Resources</i> .	38
Figura 12 – Tela da <i>Splash Activity</i>	39
Figura 13 – Tela da <i>Shader Activity</i>	39
Figura 14 – Detalhamento das classes <i>3DObject</i> e <i>Texture</i>	40
Figura 15 – Ferramenta de modelagem tridimensional	41
Figura 16 – Ordem das coordenadas de posição, normal e textura para um vértice .	41
Figura 17 – Técnica de mapeamento de textura utilizada para cada modelo 3D .	42
Figura 18 – Textura gerada a partir da técnica de mapeamento	42
Figura 19 – Detalhamento das classes <i>Timer</i> e <i>NativeLib</i>	43
Figura 20 – Detalhamento da classe <i>Renderer</i>	43
Figura 21 – Detalhamento da classe <i>Shader</i>	44
Figura 22 – <i>Shaders</i> Implementados	52
Figura 23 – Gráficos relacionados ao tempo de renderização em nanosegundos .	53
Figura 24 – Linhas de comando para execução do programa	54
Figura 25 – Diagrama de Classes do <i>script</i> de automatização	54
Figura 26 – Processo da Análise de Complexidade Algorítmica.	56
Figura 27 – Gráficos: <i>Red Shader</i> , <i>Toon shader</i> , <i>Gouraud Shader</i> , <i>Phong Shader</i> e <i>Flat Shader</i>	59
Figura 28 – Gráficos: <i>Cubemap Shader</i> , <i>Texture Shader</i> , <i>Random Shader</i> , <i>Reflection Shader</i>	60
Figura 29 – Ajustes linear, segundo, terceiro graus e exponencial para cada tipo de <i>shader</i>	61
Figura 30 – Ajustes linear, segundo, terceiro graus e exponencial para processo de renderização	62

Figura 31 – Vértices do quadrado constituído de dois triângulos	69
Figura 32 – Antes do recorte (cubo de visualização esquerdo) e depois do recorte (cubo de visualização direito)	71
Figura 33 – Travessia de triângulos: fragmentos sendo gerados	72

Lista de tabelas

Tabela 1 – Linguagens de programação para <i>shaders</i>	22
Tabela 2 – GLSL: tipos de dados	22
Tabela 3 – GLSL: qualificadores	23
Tabela 4 – Versões da plataforma <i>Android</i>	24
Tabela 5 – Valores mais comuns de complexidade algorítmica	31
Tabela 6 – Equações relacionadas ao <i>vertex shader</i> , <i>fragment shader</i>	57
Tabela 7 – Equações relacionadas ao tempo do processo de renderização	57
Tabela 8 – Exemplo de estimativa para 200000 polígonos	58
Tabela 9 – Palavras-chave do formato obj	73

Lista de abreviaturas e siglas

GPU	<i>Graphics processing unit</i>
GHz	<i>Gigahertz</i>
IDE	<i>Integrated development environment</i>
RAM	<i>Random access memory</i>
SDK	<i>Software development kit</i>
ADT	<i>Android development tools</i>
API	<i>Application Programming Interface</i>
GUI	<i>Graphical User Interface</i>
SECAM	<i>Séquentiel Couleur à Mémoire</i>
NTSC	<i>National Television System Committee</i>
RGB	<i>Red Green and Blue</i>
GLSL	<i>OpenGL Shading Language</i>
CPU	<i>Central Processing Unit</i>
NDK	<i>Native Development Kit</i>
JNI	<i>Java Native Interface</i>

Sumário

1	Introdução	17
1.1	Contextualização e Justificativa	17
1.2	Objetivos Gerais	18
1.3	Objetivos Específicos	18
1.4	Organização do Trabalho	18
2	Fundamentação Teórica	21
2.1	<i>Shaders: pipelines</i> programáveis	21
2.1.1	Linguagens dos <i>shaders</i>	22
2.1.2	Utilização dos <i>shaders</i> em <i>OpenGL</i>	22
2.1.3	<i>Shaders</i> em Plataformas Móveis	23
2.1.3.1	Plataforma <i>Android</i>	23
2.1.3.2	<i>OpenGL ES</i>	25
2.2	Fundamentação Matemática e Física para Implementação de <i>Shaders</i>	25
2.2.1	Vetor Normal e Transformações Lineares	26
2.2.2	Equação de Iluminação de <i>Phong</i>	26
2.2.3	Renderização de Efeito <i>Cartoon</i>	28
2.2.4	Renderização de Efeito de Reflexão	29
2.3	Complexidade Algorítmica Calculada de Forma Empírica	29
2.3.1	Análise Teórica	30
2.3.1.1	Notação O	31
2.3.2	Análise Experimental	31
2.3.2.1	Ajuste Linear	32
2.3.2.2	Ajuste Exponencial	32
2.3.2.3	Ajuste para Polinômio de Segundo Grau	33
2.3.2.4	Ajuste para Polinômio de Terceiro Grau	33
2.3.2.5	Cálculo dos Erros Dos Ajustes	33
3	Desenvolvimento	35
3.1	Levantamento Bibliográfico	35
3.2	Equipamentos Utilizados	35
3.3	Configuração do Ambiente	35
3.4	Implementação	37
3.4.1	Tela de <i>Front-end</i>	37
3.4.2	Objeto Tridimensional	40
3.4.3	Cálculo do Tempo de Renderização	41

3.4.4	Renderização	41
3.4.5	<i>Shaders</i>	44
3.4.5.1	<i>Phong Shader</i>	44
3.4.5.2	<i>Red Shader</i>	46
3.4.5.3	<i>Toon Shader</i>	46
3.4.5.4	<i>Simple Texture Shader</i>	47
3.4.5.5	<i>CubeMap Shader</i>	48
3.4.5.6	<i>Reflection Shader</i>	49
3.5	Análise Experimental da Complexidade Algorítmica	50
3.5.1	Medição do Tempo de Renderização Realizada pela GPU	50
3.5.2	Medição do Número de Instruções por Segundo, Plotagem e Ajuste das Curvas	50
4	Resultados	55
5	Conclusão	63
Referências	65
Anexos	67
ANEXO A Processo de Renderização	69
A.1	Processamento dos Dados dos Vértices	69
A.2	Processamento dos Vértices	70
A.3	Pós-Processamento dos Vértices	70
A.4	Montagem das Primitivas	70
A.5	Conversão e Interpolação dos Parâmetros das Primitivas	71
A.6	Processamento dos Fragmentos	71
A.7	Processamento das Amostras	71
ANEXO B Representação de Objetos Tridimensionais: Formato <i>obj</i>	73

1 Introdução

1.1 Contextualização e Justificativa

Conforme ([SHERROD, 2011](#)), os gráficos em jogos são um fator tão importante que podem determinar o seu sucesso ou fracasso. O aspecto visual é um dos pontos principais na hora da compra, juntamente com o *gameplay* (maneira que o jogador interage com o jogo). Assim, os gráficos estão progredindo na direção próxima dos efeitos visuais dos filmes, porém o poder computacional para atingir tal meta ainda tem muito a evoluir.

Neste contexto, o desempenho gráfico é um fator chave para o desempenho total de um sistema, principalmente na área de jogos, que também possui outros pontos que consomem recursos, como inteligência artificial, *networking*, áudio, detecção de eventos de entrada e resposta, física, entre outros. E isto faz com que o desenvolvimento de impressionantes efeitos visuais se tornem mais difíceis ainda.

O recente crescimento do desempenho de dispositivos móveis tornou-os capazes de suportar aplicações mais e mais complexas. Além disso, segundo ([ARNAU; PARCERISA; XEKALAKIS, 2013](#)), dispositivos como *smartphones* e *tablets* têm sido amplamente adotados, emergindo como uma das tecnologias mais rapidamente propagadas. Dentro deste contexto, a plataforma *Android*, sistema operacional *open source* para dispositivos móveis (baseado no *kernel* do *Linux*), está sendo utilizada cada vez mais, e de acordo com ([SANDBERG; ROLLINS, 2013](#)), em 2013, mais de 1,5 milhões de aparelhos utilizando esta plataforma foram ativados.

Porém, de acordo com ([NADALUTTI; CHITTARO; BUTTUSSI, 2006](#)), a renderização gráfica para dispositivos móveis ainda é um desafio devido a limitações, quando comparada a de um computador, como por exemplo, as relacionadas a CPU (*Central Processing Unit*), desempenho dos aceleradores gráficos e consumo de energia. Os autores ([ARNAU; PARCERISA; XEKALAKIS, 2013](#)) mostram que estudos prévios evidenciam que os maiores consumidores de energia em um *smartphone* são a GPU (*Graphics Processing Unit*) e a tela.

Assim, é possível analisar o desempenho do processo de renderização feito pela GPU – em que diferentes *shaders* são aplicados – por meio da complexidade algorítmica. Além disso, a análise do desempenho dos *shaders* é uma área pouco explorada, tornando o tema abordado neste trabalho mais relevante ainda.

Então, o tema consiste no desenvolvimento de *shaders* aplicados em objetos tridimensionais – com número de polígonos variável – os quais permitem a coleta de medições relacionadas ao desempenho do *device*, com relação à renderização. Desta forma, é pos-

sível variar a quantidade de polígonos de um objeto e traçar um gráfico quantidade de polígonos *versus* métrica de desempenho, utilizando um determinado *shader*. E assim, baseando-se neste gráfico, aplica-se o método dos mínimos quadrados (como explicado na Seção 2.3), a fim de determinar a complexidade algorítmica e comparar o desempenho entre os *shaders*, baseando-se na curva que melhor se ajusta à função original.

1.2 Objetivos Gerais

Os objetivos gerais do trabalho são a análise da complexidade algorítmica dos *shaders* utilizando a plataforma *Android*.

1.3 Objetivos Específicos

Os objetivos específicos do trabalho são:

- Configurar os ambientes de desenvolvimento para a plataforma *Android*;
- Implementar os *shaders* para a plataforma *Android*;
- Procurar uma ferramentas de coleta da medição de desempenho para o *device* utilizado;
- Identificar qual métrica será utilizada para a análise de complexidade;
- Coletar as medições estabelecidas;
- Automatizar o cálculo dos ajustes das curvas e plotagem dos gráficos
- Analisar a complexidade algorítmica dos *shaders*, baseado-se nas curvas obtidas.

1.4 Organização do Trabalho

No próximo capítulo serão apresentados os conceitos teóricos necessários para o entendimento do trabalho, como, por exemplo, definição de *shaders* e sua função no processo de renderização, a biblioteca gráfica utilizada, a fundamentação teórica matemática para implementação dos *shaders*, complexidade algorítmica, método dos mínimos quadrados, entre outros.

No desenvolvimento, os passos tomados no trabalho são descritos, enfatizando como foi feita a configuração do ambiente, quais equipamentos foram utilizados, como foram feitas as implementações dos *shaders* e da automatização dos cálculos e plotagem dos gráficos.

Nos resultados alcançados são descritos os resultados obtidos através da análise da complexidade algorítmica dos *shaders* implementados, seguido das conclusões do trabalho realizado.

2 Fundamentação Teórica

2.1 Shaders: pipelines programáveis

Conforme (MOLLER; HAINES; HOFFMAN, 2008), *shading* é o processo de utilizar uma equação para computar o comportamento da uma superfície de um objeto. Os *shaders* são algoritmos escritos pelo programador a fim de substituir as funcionalidades pré-definidas do processo de renderização executada pela GPU, por meio de bibliotecas gráficas como a *OpenGL*.

A *OpenGL* é uma API (*Application Programming Interface*) utilizada em computação gráfica para modelagem tridimensional, lançada em 1992, sendo uma interface de *software* para dispositivos de *hardware*. Segundo (WRIGHT et al., 2008), sua precursora foi a biblioteca Iris GL (*Integrated Raster Imaging System Graphics Library*) da empresa *Silicon Graphics*.

Antes dos *shaders* serem criados, as bibliotecas gráficas (como a *OpenGL*) possuíam um processo de renderização completamente fixo. Porém, com a introdução dos *shaders* é possível customizar parte deste processo, como é mostrada na Figura 1. O Anexo A descreve as etapas do processo ilustrado.

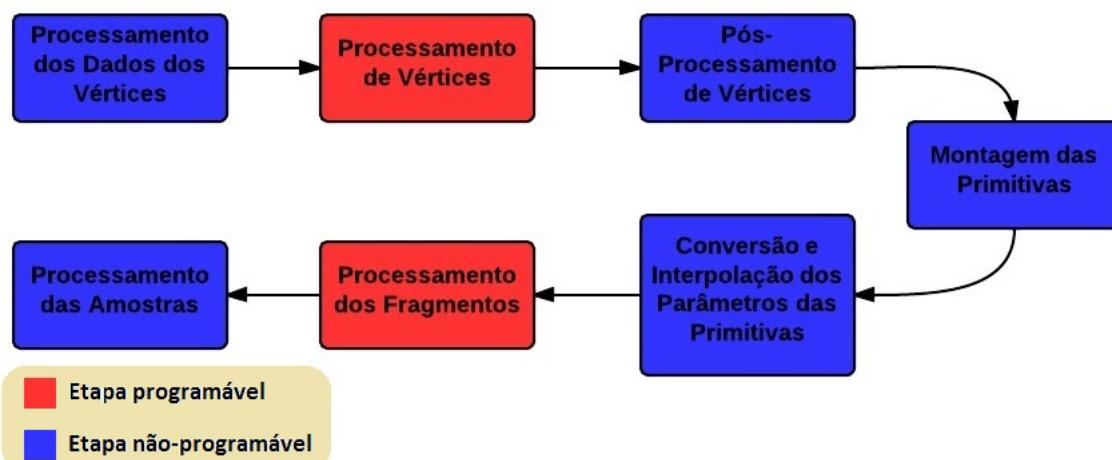


Figura 1 – Processo de renderização da *OpenGL*

Assim, existem dois tipos de *shader* principais, relacionados à criação de diferentes efeitos visuais, que focam partes distintas do *pipeline* gráfico: o *vertex shader* e o *fragment shader*. O *vertex shader* é responsável pela manipulação dos dados dos vértices, sendo responsável pela manipulação das coordenadas de posição, normal e textura, por exemplo. Ele altera a etapa de processamento de vértices e deve, ao menos, definir as coordenadas

de posição. O *fragment shader* opera na etapa de processamento dos fragmentos, em que deve atribuir uma cor para cada fragmento.

2.1.1 Linguagens dos *shaders*

Os *shaders* possuem uma linguagem de programação própria, que muitas vezes está vinculada com a API gráfica utilizada. A Tabela 1 mostra as principais linguagens de programação de *shaders* e as respectivas bibliotecas gráficas em que podem ser utilizadas.

Linguagem de Programação	Biblioteca Gráfica Suportada
<i>GLSL: OpenGL Shading Language</i>	<i>OpenGL</i>
<i>HLSL: High Level Shading Language</i>	<i>DirectX</i> e <i>XNA</i>
<i>Cg: C for Graphics</i>	<i>DirectX</i> e <i>OpenGL</i>

Tabela 1 – Linguagens de programação para *shaders*

A linguagem GLSL (*OpenGL Shading Language*) foi incluída na versão 2.0 da *OpenGL*, sendo desenvolvida com o intuito de dar aos programadores o controle de partes do processo de renderização. A GLSL é baseada na linguagem C, mas antes de sua padronização o programador tinha que escrever o código na linguagem *Assembly*, a fim de acessar os recursos da GPU. Além dos tipos clássicos do C, *float*, *int* e *bool*, a GLSL possui outros tipos mostrados na Tabela 2.

Tipo	Descrição
<code>vec2, vec3, vec4</code>	Vetores do tipo <i>float</i> de 2, 3 e 4 entradas
<code>ivec2, ivec3, ivec4</code>	Vetores do tipo inteiro de 2, 3 e 4 entradas
<code>mat2, mat3, mat4</code>	Matrizes 2x2, 3x3 e 4x4
<code>sampler1D, sampler2D, sampler3D</code>	Acesso a texturas

Tabela 2 – GLSL: tipos de dados

Além disso, a GLSL possui variáveis chamadas qualificadoras, que fazem o interfaceamento do programa e os *shaders* e entre *shaders*. Algumas destas varáveis são mostradas na Tabela 3.

2.1.2 Utilização dos *shaders* em *OpenGL*

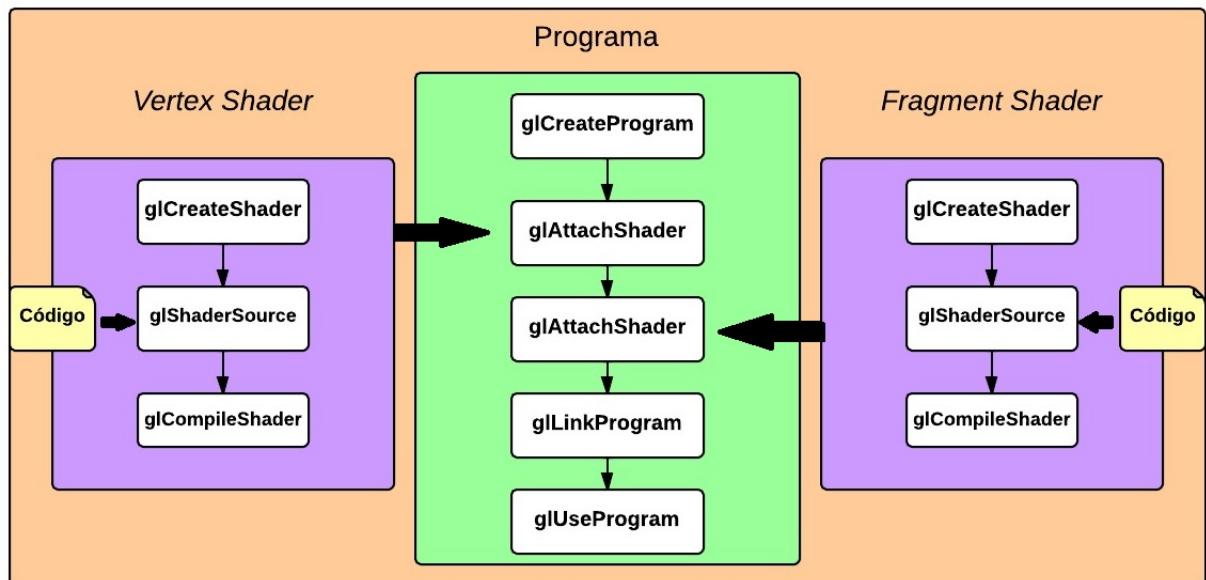
Para cada *shader* – relativos ao vértice e ao fragmento – é necessário escrever o código que será compilado e feito o *link*, gerando em um programa final que será utilizado pela *OpenGL*. Caso os *shaders* utilizem variáveis passadas pela *OpenGL*, é necessário primeiramente encontrar a localização desta variável e depois setá-la. Se a variável for do tipo *uniform*, por exemplo, adquira-se a localização por meio da função

Tipo	Descrição
attribute	Variável utilizada pelo programa para comunicar dados relacionados aos vértices para o <i>vertex shader</i>
uniform	Variável utilizada pelo programa para comunicar dados relacionados com as primitivas para ambos os <i>shaders</i>
varying	Variável utilizada pelo <i>vertex shader</i> para se comunicar com o <i>fragment shader</i>

Tabela 3 – GLSL: qualificadores

`glGetUniformLocation(GLuint program, const GLchar *name)`, em que *program* é o programa gerado a partir dos *shaders* e *name* é o nome da variável definida dentro do *shader*.

A Figura 2 mostra o processo de geração do programa a partir dos *shaders* criados.

Figura 2 – Utilização dos *shaders*

2.1.3 Shaders em Plataformas Móveis

Assim como para computadores, também existem bibliotecas gráficas para dispositivos móveis. A seguir, é explicado a plataforma *Android*, utilizada neste trabalho, e a biblioteca gráfica *OpenGL ES*, que é uma variação da *OpenGL* para dispositivos móveis, para a programação de *shaders*.

2.1.3.1 Plataforma *Android*

O *Android* começou a ser desenvolvido em 2003 na empresa de mesmo nome, fundada por Andy Rubin, a qual foi adquirida em 2005 pela empresa *Google*. A *Google* criou

a *Open Handset Alliance*, que junta várias empresas da indústria das telecomunicações, como a *Motorola* e a *Samsung*, por exemplo. Assim, elas desenvolveram o *Android* como é conhecido hoje, o qual é um sistema operacional *open source* para dispositivos móveis (baseado no *kernel* do *Linux*), tendo a primeira versão beta lançada em 2007 e segundo ([SANDBERG; ROLLINS, 2013](#)), hoje é o sistema operacional para *mobile* mais utilizado.

Ainda de acordo com ([SANDBERG; ROLLINS, 2013](#)), em 2012 mais de 3,5 *smartphones* com *Android* eram enviados aos clientes para cada *iPhone*. Em 2011, 500.000 novos *devices* eram ativados a cada dia e em 2013, os números chegam a 1,5 milhões diários. O *Android* também possui um mercado centralizado acessível por qualquer aparelho (*tablet* ou *smartphone*) chamado *Google Play*, facilitando a publicação e aquisição de aplicativos. O *Android* possui diferentes versões, sendo elas mostradas na Tabela 4 abaixo. As versões mais novas possuem mais *features* que as anteriores: a versão *Jelly Bean*, por exemplo, possui busca por voz a qual não estava disponível na versão *Ice Cream Sandwich*.

Número da versão	Nome
1.5	<i>Cupcake</i>
1.6	<i>Donut</i>
2.0/2.1	<i>Éclair</i>
2.2	<i>Fro Yo</i>
2.3	<i>Gingerbread</i>
3.0/3.1/3.2	<i>HoneyComb</i>
4.0	<i>Ice Cream Sandwich</i>
4.1/4.2/4.3	<i>Jelly Bean</i>
4.4	<i>KitKat</i>

Tabela 4 – Versões da plataforma *Android*

Uma das alternativas para o desenvolvimento em plataforma *Android* é utilizar a ferramenta *Eclipse*¹ (Figura 3), que é um ambiente de desenvolvimento integrado (*Integrated Development Environment* – IDE) *open source*. Adicionalmente, é preciso, de acordo com ([JACKSON, 2013](#)), instalar o *Android Software Development Kit*² e o *plugin ADT (Android Development Tools)*³, que permitem desenvolver e depurar aplicações pra *Android*. Outra alternativa é utilizar o *Android Studio*⁴, lançado recentemente (2013) pela empresa *Google*, que já vem com todos os pacotes e configurações necessárias para o desenvolvimento, incluindo o *Software Development Kit* (SDK), as ferramentas e os emuladores.

¹ <http://www.eclipse.org/>

² <http://developer.android.com/sdk/index.html>

³ <http://developer.android.com/tools/sdk/eclipse-adt.html>

⁴ <http://developer.android.com/sdk/installing/studio.html>

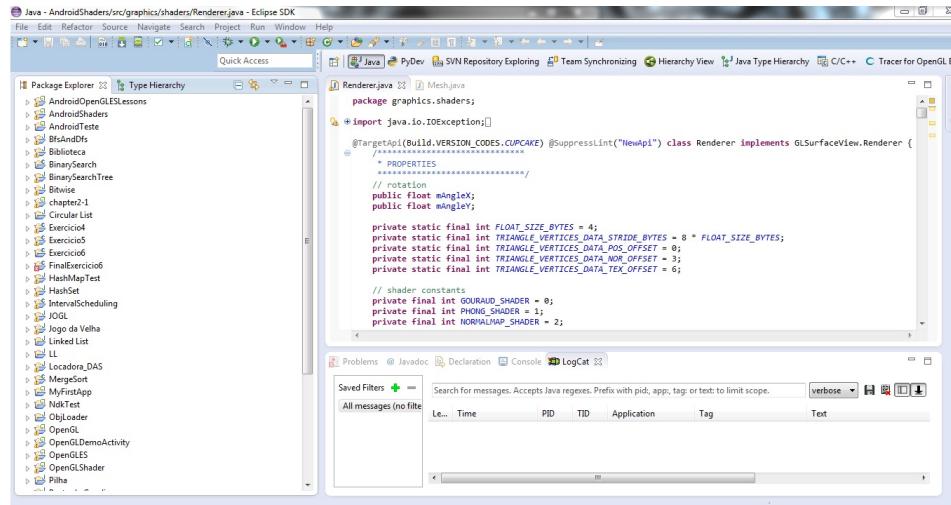


Figura 3 – Ambiente de desenvolvimento *Eclipse*

2.1.3.2 OpenGL ES

A *OpenGL ES* (*OpenGL for Embedded Systems*) foi lançada em 2003, sendo a versão da *OpenGL* para sistemas embarcados, e como citado em (GUHA, 2011), atualmente é uma das API's mais populares para programação de gráficos tridimensionais em pequenos *devices*, sendo adotada por diversas plataformas como *Android*, *iOS*, Nintendo DS e *Black Berry*.

Segundo (SMITHWICK; VERMA, 2012), ela possui três versões: a 1.x que utiliza as funções fixas de renderização, a 2.x, que elimina as funções fixas e foca nos processos de renderização manipulados por *pipelines* programáveis (*shaders*) e a 3.x, que é completamente compatível com a *OpenGL* 4.3.

A *OpenGL ES*, assim como a *OpenGL*, também utiliza a linguagem *GLSL* para programação de *shaders* e já faz parte das ferramentas de desenvolvimento da plataforma *Android*.

2.2 Fundamentação Matemática e Física para Implementação de Shaders

Os efeitos visuais – criados por meio dos *shaders* – são representações de descrições físicas, podendo atribuir materiais aos objetos e diferentes efeitos de luz, por exemplo. Assim, esta seção apresenta alguns conceitos necessários para o entendimento dos *shaders* implementados.

2.2.1 Vetor Normal e Transformações Lineares

Muitos dos *shaders* que são implementados manipulam vetores normais às superfícies dos objetos, sendo eles, vetores unitários. Porém, ao realizar diversas transformações lineares (como a de escalar por um fator), não há uma garantia que eles continuarão sendo unitários. Assim, é necessário utilizar o comando da GLSL chamado **normalize**, a fim de normalizar o vetor normal, que retorna um vetor unitário sem alterar sua direção. Porém, em uma transformação de escalar não-uniforme, por exemplo, as direções dos vetores normais são alteradas. Conserta-se isto invertendo a operação de escalar, mas torna-se complicado realizar este procedimento após sucessivas transformações lineares. Segundo ([GUHA, 2011](#)), a matriz de modelagem, visualização e projeção pode ser definida por uma série de multiplicações matriciais. Então, considerando uma matriz de modelagem resultante de duas rotações e uma operação de escalar não-uniforme, por exemplo, tem-se:

$$M_{normal} = R_1 S_{não-uniforme}^{-1} R_2 \quad (2.1)$$

Esta mesma equação pode ser reescrita de acordo com a Equação 2.2, pois invertendo uma matriz duas vezes chega-se na sua original. Além disso a inversa da matriz de rotação é igual à sua transposta e como a matriz de escalar é uma matriz diagonal, sua transposta é igual a ela mesma.

$$M_{normal} = (R_1^{-1})^T (S_{não-uniforme}^{-1})^T (R_2^{-1})^T \quad (2.2)$$

Utilizando operações de álgebra linear, a equação pode ser reescrita como:

$$M_{normal} = ((R_1 S_{não-uniforme} R_2)^{-1})^T \quad (2.3)$$

Então, a fim de consertar a direção do vetor normal, multiplica-se ele pela matriz transposta da inversa da matriz de modelagem.

2.2.2 Equação de Iluminação de *Phong*

Na área de computação gráfica, os *Flat Shading*, *Gouraud Shading* e *Phong Shading* são um dos *shaders* mais conhecidos. No método *Flat Shading*, renderiza-se cada polígono de um objeto com base no ângulo entre a normal da superfície e a direção da luz. Mesmo se as cores se diferenciem nos vértices de um mesmo polígono, somente uma cor é escolhida entre elas e é aplicada em toda o polígono.

A computação dos cálculos de luz nos vértices seguida por uma interpolação linear do resultado é conhecida como *Gouraud Shading* (considerada superior ao *Flat Shading*, pois renderiza uma superfície mais suave, lisa), criada por Henri Gouraud, sendo conhecida

como avaliação por vértice. Nela, o *vertex shader* deve calcular a intensidade em cada vértice e os resultados serão interpolados. Em seguida, o *fragment shader* pega este valor e passa adiante.

No *Phong Shading*, primeiramente interpolam-se os valores das normais das primitivas e então computam-se os cálculos de luz para cada *pixel*, utilizando as normais interpoladas. Este método também é conhecido como avaliação por *pixel*. A intensidade da luz em um ponto da superfície, segundo (GUHA, 2011), é calculada de acordo com a Equação 2.4.

$$I_{total} = I_{ambiente} + \sum_{luzes} I_{difusa} + I_{especular} \quad (2.4)$$

Assim, a intensidade de luz é calculada como a soma das intensidades ambiente, difusa (calculada para cada fonte de luz) e especular. A intensidade de reflexão ambiente vem de todas as direções e quando atinge a superfície, espalha-se igualmente em todas as direções, sendo o seu valor constante. Ela pode ser calculada de acordo com a Equação 2.5, em que K_a é o coeficiente de reflectividade ambiente da superfície e L_a é a intensidade da componente ambiente da luz.

$$I_{ambiente} = K_a L_a \quad (2.5)$$

A intensidadede reflexão difusa vem de uma direção e, assim como a ambiente, ao atingir uma superfície também espalha-se igualmente em todas as direções. Ela pode ser calculada de acordo com a Equação 2.6, em que K_d é o coeficiente de reflexão difusa da superfície, L_d é a intensidade da componente difusa da luz, \vec{l} é a fonte de luz, \vec{n} é o ponto em interesse.

$$I_{difusa} = K_d L_d (\vec{l} \cdot \vec{n}) \quad (2.6)$$

A luz especular vem de uma direção e reflete como um espelho, em que o ângulo de incidência é igual ao de reflexão, podendo ser calculada de acordo com a Equação 2.7, em que K_s é o coeficiente de reflexão especular da superfície, L_s é a intensidade da componente especular da luz, \vec{r} é a direção da reflexão, \vec{v} é o vetor de visão do ponto (observador) e s é o expoente especular.

$$I_{especular} = K_s L_s (\vec{r} \cdot \vec{v})^s \quad (2.7)$$

O *Phong Shading* requer maior poder de processamento do que a técnica *Gouraud Shading*, pois cálculos nos vértices são computacionalmente menos intensos comparados aos cálculos feitos por *pixels*. Porém, a desvantagem da técnica de *Gouraud Shading* é

que efeitos de luz que não afetam um vértice de uma superfície não surtirão efeito como, por exemplo, efeitos de luz localizados no meio de um polígono não serão renderizados corretamente. A Figura 4 mostra a diferença entre as três técnicas de *shading* aplicadas em uma esfera com uma luz direcional.

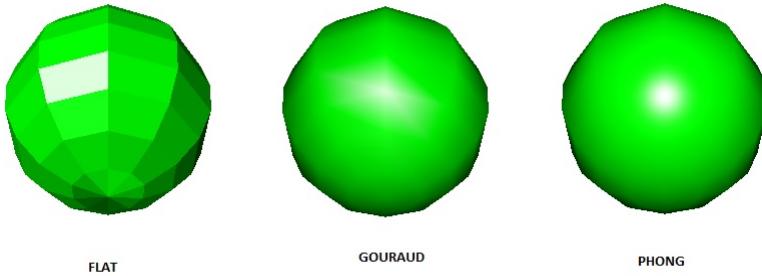


Figura 4 – Comparação entre as técnicas de *shading*

2.2.3 Renderização de Efeito *Cartoon*

A renderização de efeito *cartoon* tem como objetivo sugerir o efeito de um desenho, em que se tem como principal característica a aparência uniforme das cores sobre a superfície dos objetos. Isto pode ser feito mapeando-se faixas de intensidade de luz para tons de cores específicos, obtendo-se uma iluminação menos realista e mais próxima da utilizada em desenhos animados. De acordo com (EVANGELISTA; SILVA, 2007), o tom é escolhido baseado no cosseno do ângulo entre a direção da luz e o vetor normal da superfície. Ou seja, se o vetor normal está mais próximo da direção da luz, utiliza-se um tom mais claro. Então a intensidade da luz pode ser calculada de acordo com a Equação 2.8, em que \vec{l}_d é o vetor da direção da luz e \vec{n} é o vetor normal.

$$I_{luz} = \frac{\vec{l}_d \cdot \vec{n}}{\|\vec{l}_d\| \|\vec{n}\|} \quad (2.8)$$

Como os vetores \vec{l}_d e \vec{n} são normalizados, o cálculo pode ser simplificado de acordo com a Equação 2.9.

$$I_{luz} = \vec{l}_d \cdot \vec{n} \quad (2.9)$$

Após o cálculo da intensidade da luz, mapeia-se para os tons pré-definidos, como mostra a Figura 5

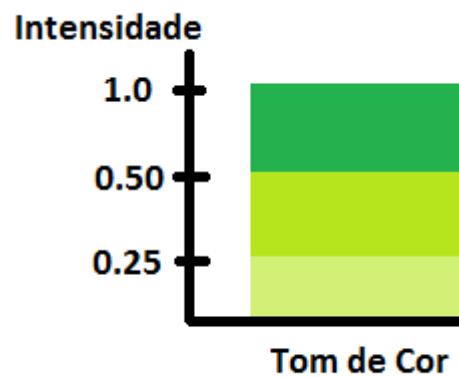


Figura 5 – Exemplo de escolha dos tons

2.2.4 Renderização de Efeito de Reflexão

O efeito de reflexão é feito através da utilização da técnica chamada *environment mapping*, em que se reflete uma textura, que é desenhada ao redor do objeto e representa o cenário, chamada de *environment map*. Um exemplo desta textura é ilustrada na Figura 6

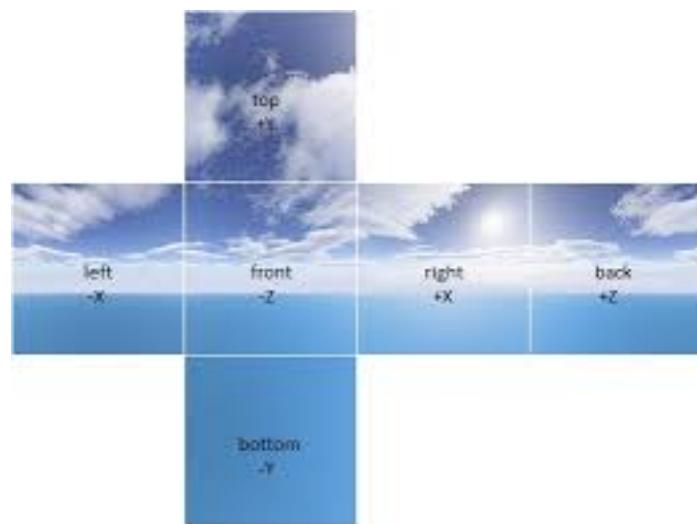


Figura 6 – *Environment Map*

Assim, a ideia é obter um vetor que vai da posição da câmera ao objeto, e após isto, ele é refletido baseando-se na normal da superfície. Isto resulta em um outro vetor que é utilizado para determinar a cor, baseando-se na textura (*environment map*).

2.3 Complexidade Algorítmica Calculada de Forma Empírica

Complexidade algorítmica é uma medida que compara a eficiência de um determinado algoritmo, analisando o quanto custoso ele é (em termos de tempo, memória, custo

ou processamento). Ela foi desenvolvida por Juris Hartmanis e Richard E. Stearns no ano de 1965. Segundo ([DROZDEK, 2002](#)), para não depender do sistema em que está sendo rodado e nem da linguagem de programação, a complexidade algorítmica se baseia em uma função (medida lógica) que expressa uma relação entre a quantidade de dados e de tempo necessário para processá-los. Assim, é possível calcular a complexidade algorítmica de um código de forma teórica e experimental.

2.3.1 Análise Teórica

O cálculo da complexidade algorítmica visa a modelagem do comportamento do desempenho do algoritmo, a medida que o número de dados aumenta. Assim, os termos que não afetam a ordem de magnitude são eliminados, gerando a aproximação denominada complexidade assintótica. Assim, a Equação (2.10) poderia ser aproximada pela Equação (2.11)

$$y = n^2 + 10n + 1000 \quad (2.10)$$

$$y \approx n^2 \quad (2.11)$$

A maioria dos algoritmos possui um parâmetro n (o número de dados a serem processados), que afeta mais significativamente o tempo de execução. De acordo com ([SEGEWICK, 1990](#)), a maioria dos algoritmos se enquadram nos tempos de execução proporcionais aos valores da Tabela 5.

A Figura 7 mostra uma comparação entre estas curvas.

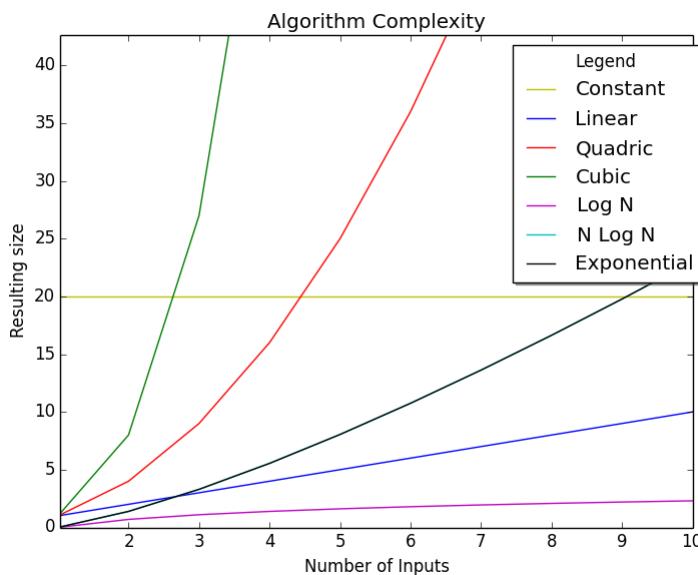


Figura 7 – Comparação da Complexidade Algorítmica

Complexidade	Descrição
Constante	Ocorre quando as instruções do programa independe do número de elementos de entrada. As instruções são executadas em um número fixo de vezes.
$\log N$	Ocorre geralmente em programas que resolvem grandes problemas dividindo-os em partes menores, cortando o seu tamanho por uma constante.
N	Ocorre quando o programa é linear, ou seja, o processamento é feito para cada elemento de entrada.
$N \log N$	Ocorre quando o problema é quebrado em partes menores, sendo resolvidas independentemente, e depois suas soluções são combinadas
N^2	Ocorre quando o algoritmo é quadrático, ou seja, quando processa todos os pares de itens de dados.
N^3	Ocorre quando o algoritmo é cúbico, ou seja, quando processa todos as tripas de itens de dados.
2^N	Ocorre quando o algoritmo segue uma função exponencial, ou seja, quando o N dobra o tempo de execução vai ao quadrado.

Tabela 5 – Valores mais comuns de complexidade algorítmica

2.3.1.1 Notação O

A notação O foi desenvolvida em 1894 por Paul Bachmann para complexidade assintótica. Assim, segundo (DROZDEK, 2002), dadas duas funções de valores positivos f e g , $f(n)$ é $O(g(n))$ se existem c e N positivos tais que $f(n) \leq cg(n)$, $\forall n \geq N$. Ou seja, dada uma função $g(n)$, denota-se por $O(g(n))$ o conjunto das funções que para valores de n suficientemente grandes, $f(n)$ é igual ou menor que $g(n)$. A função f tende a crescer no máximo tão rápido quanto g , em que $cg(n)$ é uma cota superior de $f(n)$. O algoritmo $y = n^2 + 10n + 1000$, por exemplo, é igual a $O(n^2)$.

Se $f(n)$ é um polinômio de grau d então $f(n)$ é $O(n^d)$. E como uma constante pode ser considerada de grau zero, sua complexidade é $O(n^0)$, ou seja, $O(1)$. Além disso, a função $f(n) = \log_a(n)$ é $O(\log_b(n))$ para quaisquer a e b positivos diferentes de 1.

2.3.2 Análise Experimental

Como não é possível analisar a complexidade algorítmica dos *shaders* de forma teórica, pois não é possível saber como é feita a implementação das funções disponíveis pela GLSL, uma das formas de calculá-la é de maneira experimental. Isto é feito variando o número de entradas e coletando uma medição associada ao desempenho, como o tempo, por exemplo. Assim, é possível gerar um gráfico e utilizar o método dos mínimos quadrados para descobrir a qual curva estes pontos pertencem. O método dos mínimos quadrados é

usado para ajustar pontos (x, y) determinados experimentalmente a uma curva.

2.3.2.1 Ajuste Linear

No caso do ajuste a uma reta (dada por $y = a + bx$), por exemplo, muitas vezes os pontos não são colineares e segundo (RORRES, 2001) é impossível encontrar coeficientes a e b que satisfaçam o sistema. Então, as distâncias destes valores para a reta podem ser consideradas como medidas de erro e os pontos são minimizados pelo mesmo vetor (minimizando a soma dos quadrados destes erros). Assim, existe um ajuste linear de mínimos quadrados aos dados, e a sua solução é dada pela Equação (2.12), em que é possível determinar os coeficientes a e b e consequentemente, a equação da reta.

$$v = (M^T M)^{-1} M^T y \quad (2.12)$$

onde

$$M = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ . & . \\ . & . \\ 1 & x_n \end{bmatrix}, \quad v = \begin{bmatrix} a \\ b \end{bmatrix} \text{ e } y = \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ y_n \end{bmatrix} \quad (2.13)$$

2.3.2.2 Ajuste Exponencial

De acordo com (LEITHOLD, 1994), a função da exponencial pode ser dada como na Equação (2.14), em que e , c , k são constantes (e é a constante neperiana).

$$y = ce^{-kt} \quad (2.14)$$

Aplicando a função logarítmico dos dois lados da equação, obtém-se a Equação (2.15)

$$\ln y = \ln c + \ln e^{-kt} \quad (2.15)$$

que pode ser simplificada na Equação (2.16) (em que b é uma nova constante) que equivale à equação da reta.

$$\bar{y} = \bar{a} + \bar{b}t \quad (2.16)$$

Assim, é possível aplicar os métodos dos mínimos quadrados descritos anteriormente, aplicando o logaritmo nos dois lados da equação da exponencial. Os novos valores de M e y passam a ser:

$$M = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \cdot & \cdot \\ \cdot & \cdot \\ 1 & x_n \end{bmatrix}, \quad y = \begin{bmatrix} \ln y_1 \\ \ln y_2 \\ \cdot \\ \cdot \\ \ln y_n \end{bmatrix} \quad (2.17)$$

O valores finais dos coeficientes \bar{a} e \bar{b} determinam os parâmetros c e k da exponencial através das relações

$$c = e^{\bar{a}} \text{ e } \bar{b} = -k \quad (2.18)$$

2.3.2.3 Ajuste para Polinômio de Segundo Grau

O ajuste de segundo grau é parecido com o linear, em que a Equação 2.12 também é utilizada. Porém a matriz M é redefinida para:

$$M = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ 1 & x_n & x_n^2 \end{bmatrix}, \quad (2.19)$$

2.3.2.4 Ajuste para Polinômio de Terceiro Grau

O ajuste de segundo grau é ocorre da mesma forma que o de segundo grau, mas a matriz M é redefinida para:

$$M = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ 1 & x_n & x_n^2 & x_n^3 \end{bmatrix}, \quad (2.20)$$

2.3.2.5 Cálculo dos Erros Dos Ajustes

O cálculo dos erros dos ajustes para as funções é calculado a fim de saber qual delas mais se aproxima da curva original. Estes erros são calculados de acordo com a Equação 2.21, em que cada erro (e_n) está associado à distância da coordenada da função ajustada para a função original.

$$E = \sqrt[3]{e_1 + e_2 + \dots + e_n} \quad (2.21)$$

3 Desenvolvimento

Este capítulo descreve a metodologia adotada para a realização deste trabalho, mostrando desde os equipamentos utilizados até como foram feitos os procedimentos de implementação, coleta e análise dos dados.

3.1 Levantamento Bibliográfico

Uma vez escolhida a área de interesse e o tema, a primeira etapa do trabalho consistiu em um levantamento bibliográfico, a fim de avaliar a disponibilidade de material para fomentar o tema do trabalho e também analisar o que já foi desenvolvido na área. Feito isso, tomando-se como base o que já foi publicado e desenvolvido, foram definidas as possíveis contribuições, identificando (como foi dito no Capítulo 1) a limitação de desempenho e o desenvolvimento para *mobile* como áreas a serem exploradas.

3.2 Equipamentos Utilizados

Uma vez definidos os temas e as limitações do trabalho, foram escolhidos os equipamentos que seriam utilizados no desenvolvimento do trabalho. O celular utilizado foi o *Nexus 4*, o qual é o quarto *smartphone* da *Google*, projetado e fabricado pela *LG Electronics*. Ele possui o processador *Snapdragon S4 Pro* de 1,512 GHz *quad-core*, GPU (*Graphics processing unit*) *Adreno 320* e 2 GB de memória RAM. O computador utilizado na codificação e nos testes foi o da linha *Alienware M14x* fabricado pela *Dell*, no qual possui processador *Intel Core i7* de 2,3 GHz, GPU *NVIDIA GeForce GTX* de 2 GB e 8 GB de memória RAM.

3.3 Configuração do Ambiente

Em seguida, foram feitas as configurações dos ambientes de trabalho. Para desenvolver na plataforma *Android* foi necessário instalar o *Android SDK*, *Android NDK* (para utilizar linguagem de código nativo C) e o *plugin ADT*, uma vez que seria utilizada a IDE *Eclipse*. A biblioteca gráfica para sistemas embarcados *OpenGL ES* faz parte das ferramentas de desenvolvimento da plataforma *Android*.

Para a coleta de métricas foi utilizada a ferramenta *Adreno Profiler*, pois o celular utilizado possui a GPU *Adreno*. A *Adreno Profiler* é uma ferramenta que foca na otimização gráfica para celulares que possuem GPU *Adreno* (fabricada pela empresa *Qualcomm*).

De acordo com (QUALCOMM, 2013), a ferramenta provê suporte para *Android* e *Windows RT* (variação do sistema operacional *Windows 8* e projetada para *devices* móveis), permitindo a otimização, análise por quadros e visualização de desempenho em tempo real. No dispositivo *Nexus 4* ela só é suportada com o *Android* até a versão 4.3 (não suporta o *KitKat*).

Como pode ser visto na Figura 8, a ferramenta possui um módulo de análise dos *vertex* e *fragment shaders*, sendo possível editá-los e analisar os resultados da compilação em tempo real, além dela também gerar estatísticas.

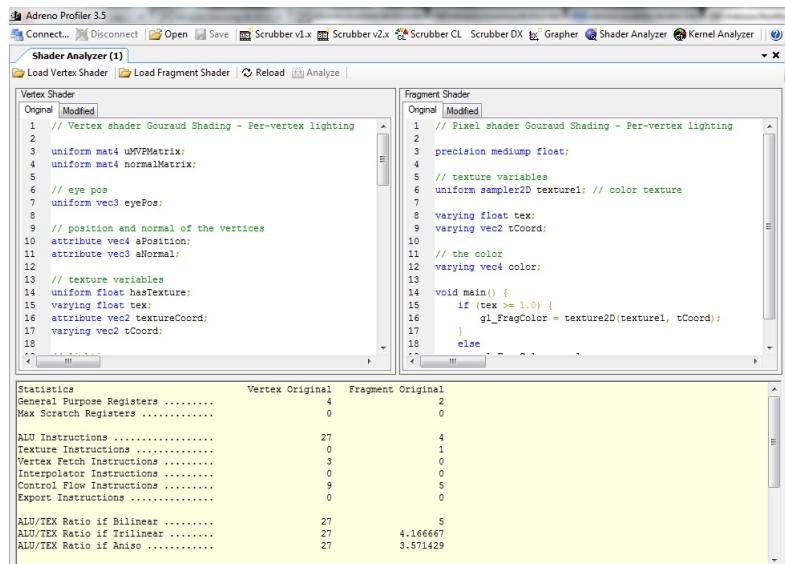


Figura 8 – Ferramenta *Adreno Profiler*: analisador de *shaders*

O módulo gráfico permite analisar algumas métricas, relacionadas ao *vertex* e *fragment shaders*, conforme ilustrado na Figura 9, onde um gráfico é plotado em tempo de execução. Além disso, ela também exporta os resultados no formato CSV (*Comma-Separated Values*), que consiste em um arquivo de texto que armazena valores tabelados separados por um delimitador (vírgula ou quebra de linha). O último módulo é o chamado *Scrubber*, que provê informações detalhadas quanto ao rastreamento de uma chamada de função.

Para a automatização da plotagem dos gráficos e aplicação do método dos mínimos quadrados (descrito na Seção 2.3.2) utilizou-se a linguagem de programação Python¹ versão 2.7.3, juntamente com os pacotes *matplotlib*² e *numpy*³.

O controle de versionamento do código foi feito por meio do sistema de controle de versão Git⁴ utilizando o *forge GitHub*⁵. Foram criados repositórios públicos tanto para

¹ <http://www.python.org.br/>

² <http://www.matplotlib.org/>

³ <http://www.numpy.org/>

⁴ <http://git-scm.com/>

⁵ <https://github.com/>



Figura 9 – Ferramenta *Adreno Profiler*: visualização de métrica quadros por segundo

a implementação em *Android*⁶ quanto para a automatização da análise da complexidade algorítmica⁷.

3.4 Implementação

A fim de tornar possível a realização da análise da complexidade algorítmica experimentalmente, primeiramente focou-se na implementação dos *shaders* para plataforma *Android* utilizando a biblioteca *OpenGL ES*. Foi utilizado o paradigma de orientação a objetos, em que o diagrama de classes (Figura 10) mostra como o código foi estruturado. Este diagrama mostra um conjunto de classes e seus relacionamentos, sendo o diagrama central da modelagem orientada a objetos.

3.4.1 Tela de *Front-end*

A tela de *front-end* é responsável pela interação com o usuário, repassando as informações de entrada para o *back-end*. E de acordo com (JACKSON, 2013), a plataforma *Android* utiliza o termo *Activity* para descrever esta tela de *front-end* da aplicação. Ela possui elementos de *design* como texto, botões, gráficos, entre outros. No contexto deste trabalho, há duas classes *Activity* (Figura 11), a *Shader* e a *Splash*.

A *Splash Activity* (Figura 12) é responsável pela visualização da tela de *loading* enquanto carrega os recursos necessários para o programa (como a leitura dos modelos tridimensionais em formato *obj* e das imagens usadas para texturização) por meio do uso de *thread*. Estes recursos são gerenciados pela classe *Resources*, que utiliza o padrão de

⁶ https://github.com/campeloal/monografia_opengles

⁷ https://github.com/campeloal/monografia_algorithmComplexity/

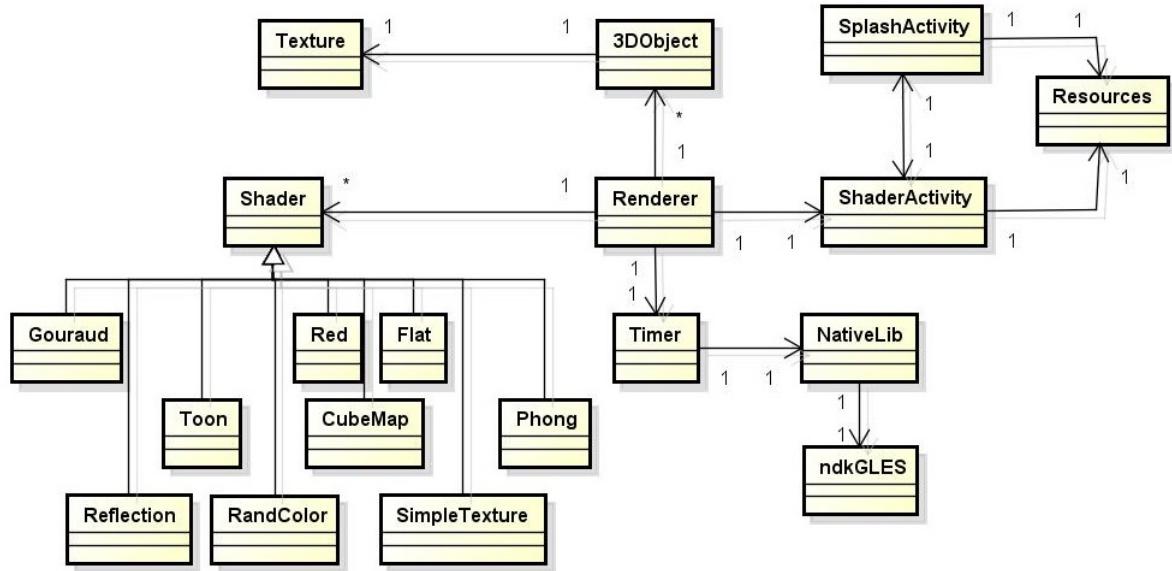


Figura 10 – Diagrama de Classe da Implementação em *Android*

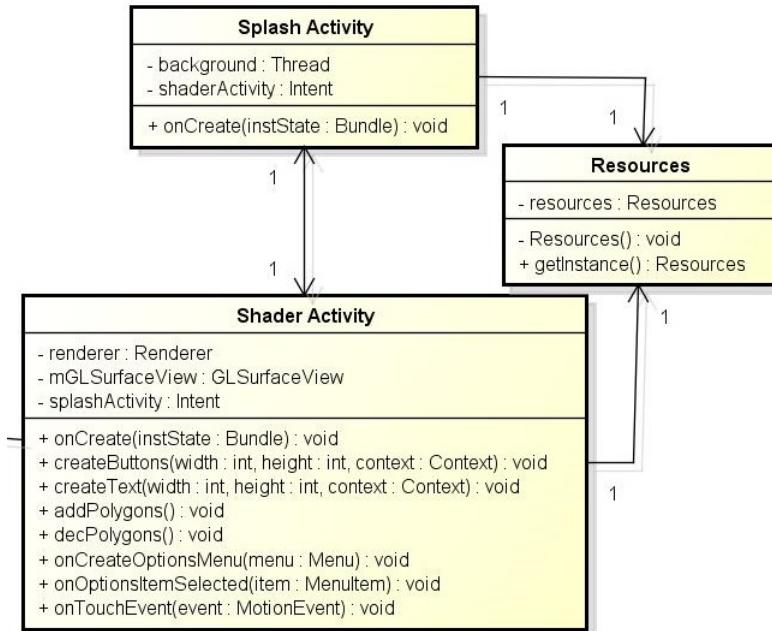


Figura 11 – Detalhamento das classes *Shader Activity*, *Splash Activity* e *Resources*

projeto *Singleton*, que garante a existência de apenas uma instância da classe, que será acessada posteriormente.

A *Shader Activity* (Figura 13) é responsável pela instanciação da classe *Renderer*, que renderiza os gráficos tridimensionais utilizando a biblioteca *OpenGL ES*. Além disso, ela controla os eventos de *touch*, que permitem escalar e mover o objeto, além de disponibilizar os menus que trocam de *shaders*, os botões que aumentam ou diminuem o número de polígonos, a informação do tempo de renderização e a de quantidade de polígonos. O

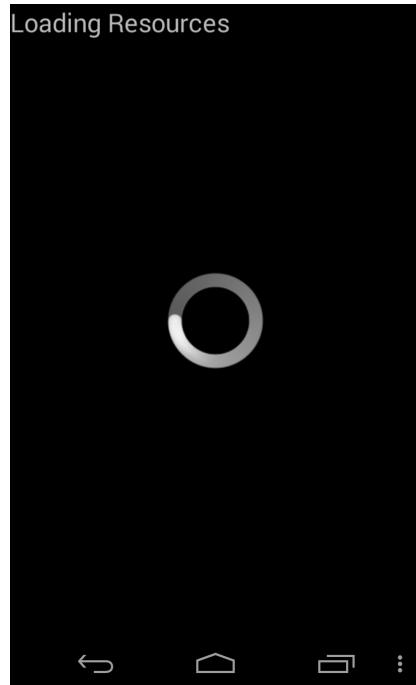


Figura 12 – Tela da *Splash Activity*

aumento do número de polígonos é realizado através da troca de objetos que possuem arquivos *obj* diferentes, que já foram carregados pela *Splash Activity*.



Figura 13 – Tela da *Shader Activity*

Devido à limitação de memória do dispositivo móvel e os vários objetos com diferentes números de polígonos, não é possível carregar todos de uma só vez. Assim, foi necessário delimitar esta quantidade de objetos simultâneos: uma vez atingido o valor

limítrofe (tanto adicionando, quanto decrementando), volta-se novamente para a *Splash Activity*, a fim de carregar os novos objetos e retornar para a *Shader Activity*, onde serão renderizadas.

3.4.2 Objeto Tridimensional

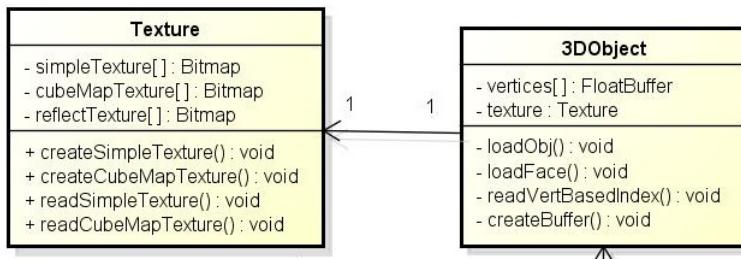


Figura 14 – Detalhamento das classes *3DObject* e *Texture*

O objeto tridimensional foi representado pela composição das classes *3DObject* e *Texture*. A classe *3DObject*, mostrada na Figura 14, é responsável por ler e interpretar os arquivos *obj* (descrito no Anexo B). Estes arquivos foram criados e exportados utilizando a ferramenta de modelagem tridimensional *open source* chamada Blender⁸, a qual está ilustrada na Figura 15. Ela também permite modificar o número de polígonos de um modelo tridimensional por meio de modificadores chamados *Decimate* e *Subdivision Surface* (usados para diminuir e aumentar a contagem poligonal, respectivamente). Assim, foi possível variar a contagem poligonal a partir de diferentes arquivos *obj*.

Após a leitura e interpretação do arquivo *obj* foi gerado um *buffer* para armazenar os vértices de posição, normal e textura na ordem em que eles serão renderizados. Neste *buffer* cada coordenada relacionada a um vértice (posição, normal e textura) é armazenada alternadamente, como mostra a Figura 16.

A classe *Texture* gera as texturas utilizadas pelos *shaders SimpleTexture*, *Cube-Map* e *Reflection* baseando-se em imagens. As imagens são criadas para cada modelo tridimensional, utilizando a técnica de *UV Mapping*, na qual mapeiam-se as coordenadas de textura para uma imagem (Figura 17). Como a orientação do eixo de coordenadas *y* da ferramenta *Blender* é diferente da *OpenG ES*, é necessário refletir a imagem neste eixo para corrigir o mapeamento.

Para gerar uma textura simples, primeiramente gera-se um objeto de textura utilizando a função `glGenTextures`, depois vincula-se esta textura ao objeto com a função `glBindTexture` e carrega-se a imagem por meio da função `texImage2D`. Para as texturas do *CubeMap* e *Reflection*, faz-se a mesma coisa, exceto que a função `texImage2D` é feita seis vezes, uma vez que cada textura representa uma face de um cubo.

⁸ <http://www.blender.org/>



Figura 15 – Ferramenta de modelagem tridimensional

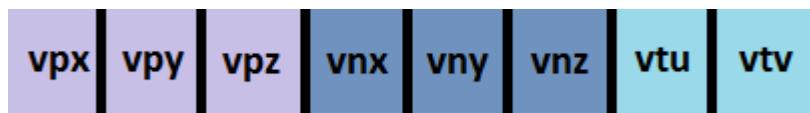


Figura 16 – Ordem das coordenadas de posição, normal e textura para um vértice

Na Figura 18 é mostrada a imagem resultante da técnica de mapeamento realizada.

3.4.3 Cálculo do Tempo de Renderização

A Figura 19 detalha a classe *Timer*, que realiza a média de dez medições do tempo de renderização (em nanosegundos) para cada objeto tridimensional, utilizando um *shader* específico. Cada medição é feita utilizando a linguagem C e a extensão de *OpenGL ES* chamada *GL_EXT_disjoint_timer_query* citada na Seção 3.5.1. A integração entre o código em linguagem C e o código em Java é feita por meio da classe *NativeLib*.

3.4.4 Renderização

A Figura 20 mostra a classe *Renderer*, responsável pela renderização, que funciona como uma controladora, sendo o ponto principal das chamadas provenientes da *view* (*ShaderActivity*) para as classes de *model* (*3DObject*, *Shader* e *Timer*). Ela que

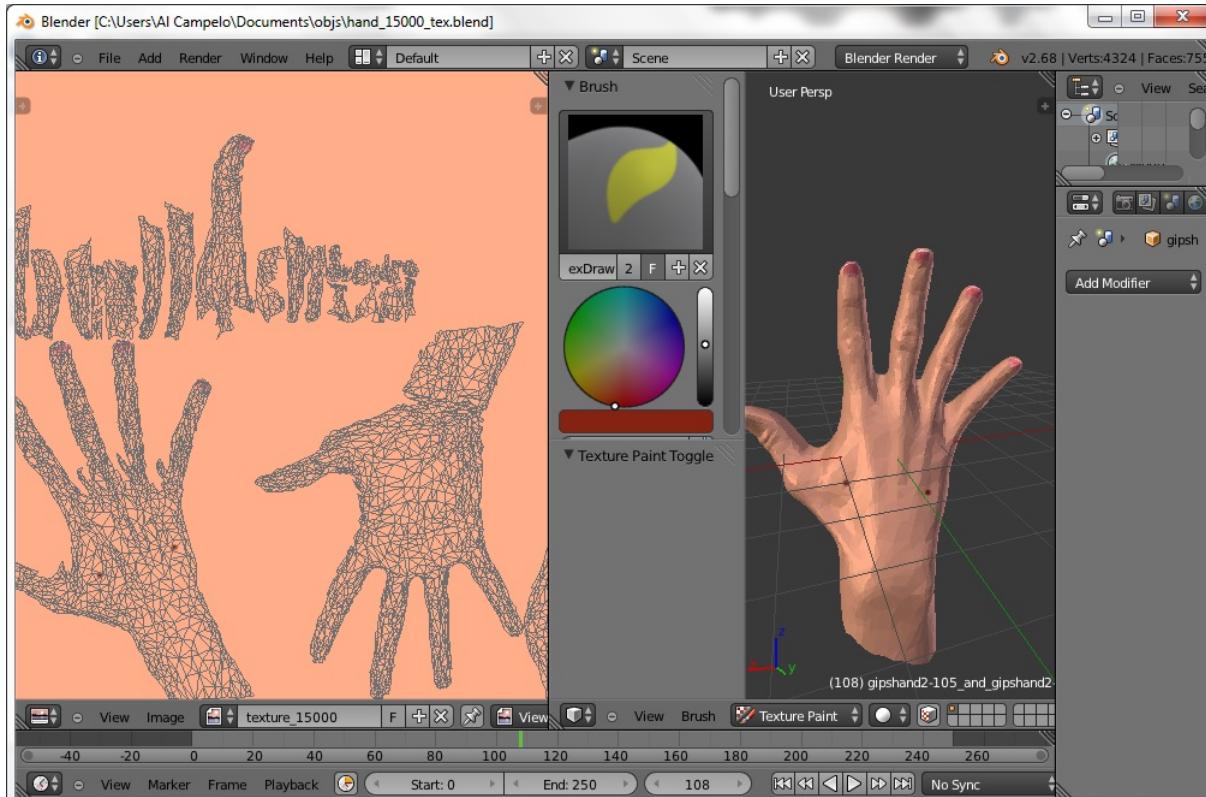
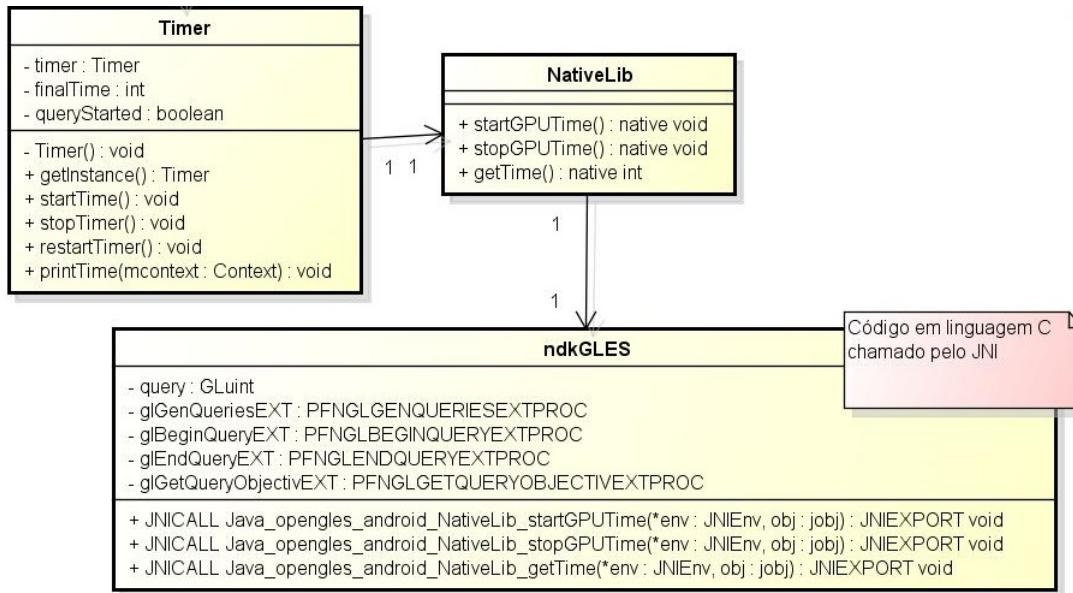
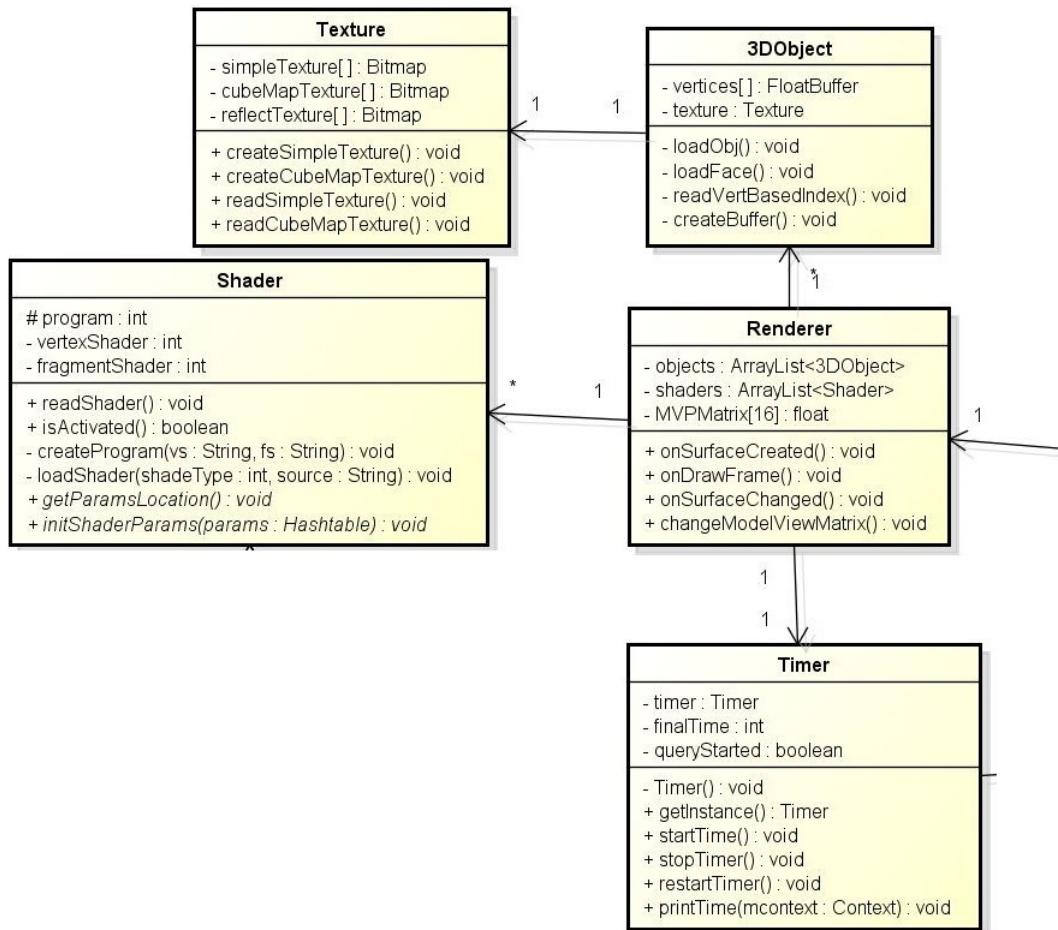


Figura 17 – Técnica de mapeamento de textura utilizada para cada modelo 3D



Figura 18 – Textura gerada a partir da técnica de mapeamento

implementa as funções da biblioteca *OpenGL ES* `onSurfaceCreated`, `onDrawFrame` e `onSurfaceChanged`. A primeira função é chamada apenas uma vez quando a *view* da *OpenGL ES* é instanciada, e é responsável por todas as configurações, como por exemplo, a criação de texturas. A segunda função é chamada em *loop*, em que é feita a renderização por meio da função `glDrawArrays`.

Figura 19 – Detalhamento das classes *Timer* e *NativeLib*Figura 20 – Detalhamento da classe *Renderer*

3.4.5 Shaders

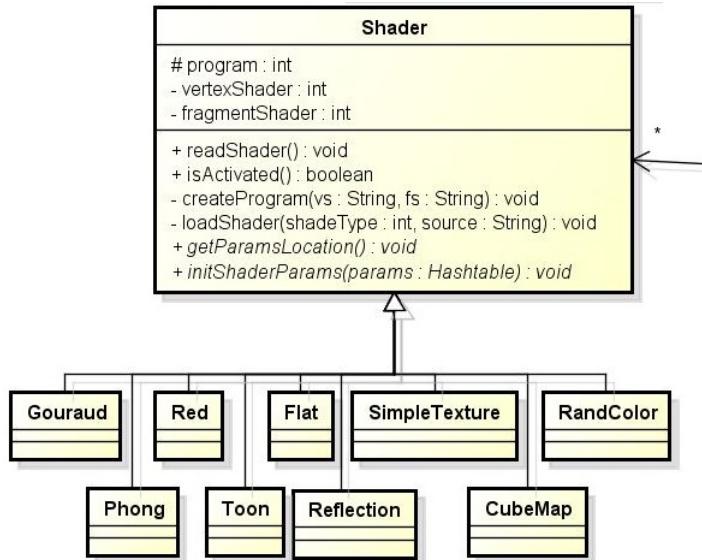


Figura 21 – Detalhamento da classe *Shader*

A classe *Shader* (Figura 21) é responsável por ler, fazer o *attach* e o *link* do *vertex* e do *fragment shaders*. Além disso, ela possui os métodos abstratos `getParamsLocation` e `initShaderParams(Hashtable params)`. Assim, todos os *shaders* que herdam desta classe são obrigados a implementar estes métodos. O primeiro método faz o armazenamento da localização de cada variável especificada dentro do *shader*, já o segundo método inicializa estas variáveis por meio de um *hash* que é passado como um parâmetro pela classe *Renderer*. Assim, todos os *shaders* implementados herdam da classe *Shader* e implementam seus métodos abstratos. Estes *shaders* podem ser vistos na Figura 22.

A seguir, alguns dos *shaders* implementados serão apresentados:

3.4.5.1 Phong Shader

O *vertex* e *fragment shaders* do *phong shading* implementam a técnica descrita na Seção 2.2.2, em que primeiramente interpolam-se os valores dos vetores normais das primitivas e então computam-se os cálculos de luz para cada fragmento, utilizando os normais interpolados. O Código 3.1 e o Código 3.2 mostram as definições do *vertex* e *fragment shaders*, respectivamente, os quais utilizam as variáveis que definem as propriedades do material.

Código 3.1 – *Phong Shader: vertex shader*

```

1 uniform mat4 uMVPMatrix;
2 uniform mat4 normalMatrix;
3 uniform vec3 eyePos;
4 attribute vec4 aPosition;
  
```

```

5  attribute vec3 aNormal;
6  uniform vec4 lightPos;
7  uniform vec4 lightColor;
8  uniform vec4 matAmbient;
9  uniform vec4 matDiffuse;
10 uniform vec4 matSpecular;
11 uniform float matShininess;
12 varying vec3 vNormal;
13 varying vec3 EyespaceNormal;
14 varying vec3 lightDir, eyeVec;
15
16 void main() {
17
18     EyespaceNormal = vec3(normalMatrix * vec4(aNormal, 1.0));
19     vec4 position = uMVPMatrix * aPosition;
20     lightDir = lightPos.xyz - position.xyz;
21     eyeVec = -position.xyz;
22
23     gl_Position = uMVPMatrix * aPosition;
24 }
```

Código 3.2 – *Phong Shader: fragment shader*

```

1 precision mediump float;
2 varying vec3 vNormal;
3 varying vec3 EyespaceNormal;
4 uniform vec4 lightPos;
5 uniform vec4 lightColor;
6 uniform vec4 matAmbient;
7 uniform vec4 matDiffuse;
8 uniform vec4 matSpecular;
9 uniform float matShininess;
10 uniform vec3 eyePos;
11 varying vec3 lightDir, eyeVec;
12
13 void main() {
14     vec3 N = normalize(EyespaceNormal);
15     vec3 E = normalize(eyeVec);
16     vec3 L = normalize(lightDir);
17     vec3 reflectV = reflect(-L, N);
18     vec4 ambientTerm;
19     ambientTerm = matAmbient * lightColor;
20     vec4 diffuseTerm = matDiffuse * max(dot(N, L), 0.0);
```

```

21     vec4 specularTerm = matSpecular *
22         pow(max(dot(reflectV, E), 0.0), matShininess);
23
24     gl_FragColor = ambientTerm + diffuseTerm + specularTerm;
25
26 }
```

3.4.5.2 Red Shader

O *shader* que define a cor do fragmento como vermelha é muito simples: o *vertex shader* apenas estabelece que a posição do vértice se dá pela multiplicação da matriz de projeção, visualização e modelagem pela coordenada (variável `aPosition`) como é mostrada no Código 3.3.

Código 3.3 – Red Shader: *vertex shader*

```

1 uniform mat4 uMVPMatrix;
2 attribute vec4 aPosition;
3
4 void main() {
5     gl_Position = uMVPMatrix * aPosition;
6 }
```

Já o seu *fragment shader* (Código 3.4) estabelece que todo fragmento possui a cor vermelha, por meio da variável pré-definida `gl_FragColor`.

Código 3.4 – Red Shader: *fragment shader*

```

1 void main() {
2     gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
3 }
```

3.4.5.3 Toon Shader

O *toon shader* calcula a intensidade da luz por vértice para escolher uma das cores pré-definidas, como apresentado na Seção 2.2. O Código 3.5 mostra o cálculo da intensidade da luz por vértice, pegando primeiro a direção da luz (definida como uma variável `uniform` passada pelo programa) para depois fazer o produto escalar entre ela e o vetor normal (cálculo da intensidade da luz).

Código 3.5 – Toon Shader: *vertex shader*

```

1 uniform vec3 lightDir;
2 uniform mat4 uMVPMatrix;
3 attribute vec3 aNormal;
4 attribute vec4 aPosition;
5 varying float intensity;
```

```

6
7 void main()
8 {
9     intensity = dot(lightDir, aNormal);
10    gl_Position = uMVPMatrix * aPosition;
11 }

```

A variável *intensity* do tipo *varying* é passada do *vertex shader* para o *fragment shader*, a fim de determinar qual das três cores será escolhida (Código 3.6).

Código 3.6 – *Toon Shader: fragment shader*

```

1 varying float intensity;
2
3 void main()
4 {
5     vec4 color;
6
7     if (intensity > 0.95)
8         color = vec4(0.5, 1.0, 0.5, 1.0);
9     else if (intensity > 0.5)
10        color = vec4(0.3, 0.6, 0.3, 1.0);
11    else
12        color = vec4(0.1, 0.2, 0.1, 1.0);
13
14    gl_FragColor = color;
15 }

```

3.4.5.4 Simple Texture Shader

O *vertex shader* do *simple texture shading* primeiramente armazena as coordenadas de textura numa variável do tipo *varying* (Código 3.7), e as repassa para o *fragment shader*, além de também definir a posição do vértice.

Código 3.7 – *Simple Texture Shader: vertex shader*

```

1 uniform mat4 uMVPMatrix;
2 attribute vec4 aPosition;
3 attribute vec2 textCoord;
4 varying vec2 tCoord;
5
6 void main() {
7     tCoord = textCoord;
8     gl_Position = uMVPMatrix * aPosition;
9 }

```

No Código 3.8, o *fragment shader*, por sua vez, utiliza a textura passada pelo programa e aplica a coordenada de textura, repassada pelo *vertex shader*, no fragmento por meio da função `texture2D`.

Código 3.8 – *Simple Texture Shader: fragment shader*

```

1 varying vec2 tCoord;
2 uniform sampler2D texture;
3
4 void main()
5 {
6     gl_FragColor = texture2D(texture, tCoord);
7 }
```

3.4.5.5 CubeMap Shader

O *vertex shader* do *CubeMap Shader* é simples e só define a posição do vértice (Código 3.9).

Código 3.9 – *CubeMap Shader: vertex shader*

```

1 attribute vec4 aPosition;
2 attribute vec3 aNormal;
3 varying vec3 v_normal;
4 uniform mat4 uMVPMatrix;
5
6 void main()
7 {
8     gl_Position = uMVPMatrix * aPosition;
9     v_normal = aNormal;
10 }
```

O *fragment shader* (Código 3.10), por sua vez, utiliza a função `textureCube` que recebe como parâmetro o vetor normal para fazer o mapeamento da textura.

Código 3.10 – *CubeMap Shader: fragment shader*

```

1 precision mediump float;
2 varying vec3 v_normal;
3 uniform samplerCube s_texture;
4 void main()
5 {
6     gl_FragColor = textureCube( s_texture, v_normal );
7 }
```

3.4.5.6 Reflection Shader

O *Reflection Shader* implementa a técnica descrita na Seção 2.2.4. O seu *vertex shader* é responsável por indicar que a posição do vértice se dá pela multiplicação da coordenada obtida pela variável `vec4 aPosition` com a matriz de projeção, visualização e modelagem, como é mostrado no Código 3.11. Além disso, ele declara dois vetores do tipo *varying* (para passar ao *fragment shader*) que estão relacionados com o vetor de direção da câmera com o vetor normal.

Código 3.11 – *Reflection Shader: vertex shader*

```

1 attribute vec4 aPosition;
2 attribute vec3 aNormal;
3
4 varying vec3 EyeDir;
5 varying vec3 Normal;
6
7 uniform mat4 MVMatrix;
8 uniform mat4 uMVPMatrix;
9 uniform mat4 NMatrix;
10
11 void main()
12 {
13     gl_Position = uMVPMatrix * aPosition;
14     EyeDir=vec3(MVMatrix*aPosition);
15     Normal = mat3(NMatrix) * aNormal;
16 }
```

No *fragment shader*, a normal e o vetor de direção da câmera são utilizados para encontrar o vetor da direção da reflexão através da utilização da função `reflect`. Para corrigir a direção da normal, ela é multiplicada pela transposta da inversa da matriz de projeção, modelagem e visualização, como foi também explicado na Seção 2.2. O vetor da direção da reflexão é utilizado na função `textureCube` (Código 3.12), em que determina-se a cor do fragmento, baseando-se nesta direção e em uma imagem.

Código 3.12 – *Reflection Shader: fragment shader*

```

1 varying vec3 EyeDir;
2 varying vec3 Normal;
3 uniform samplerCube s_texture;
4
5 void main()
6 {
7     reflectedDirection.y = -reflectedDirection.y;
8     gl_FragColor = textureCube( s_texture, reflectedDirection);}
```

3.5 Análise Experimental da Complexidade Algorítmica

A análise experimental da complexidade algorítmica foi através da coleta de diversas medições para cada modelo tridimensional (com diferentes quantidades de polígonos). Estas medidas foram plotadas em gráficos, cujas curvas foram ajustadas através do método dos mínimos quadrados (Seção 2.3.2).

3.5.1 Medição do Tempo de Renderização Realizada pela GPU

A fim de estimar experimentalmente a análise da complexidade algorítmica dos *shaders* implementados, primeiro buscou-se coletar uma métrica relacionada ao tempo de renderização. Por meio de pesquisa e consulta na documentação da *OpenGL ES*, conseguiu-se encontrar uma extensão desta biblioteca gráfica que permite contabilizar o tempo, em nanosegundos, necessário para realizar chamadas de *OpenGL ES* específicas. Esta extensão se chama `GL_EXT_disjoint_timer_query`⁹ e só está disponível para o dispositivo *Nexus 4* a partir da versão de Android 4.4 (*KitKat*).

Para utilizar esta extensão foi necessário instalar e configurar o NDK (*Native Development Kit*) e alterar o *header* da versão da *OpenGL ES* utilizada, adicionando as linhas de código relacionadas à extensão almejada. Assim, foi possível utilizar as novas funções desta extensão pegando os seus endereços por meio do comando `eglGetProcAddress` (disponível por meio da API EGL que faz o interfaceamento entre a *OpenGL ES* e o sistema operacional). A integração entre o código em linguagem C e o código em Java foi feita por meio da JNI (*Java Native Interface*).

Feita a configuração e implementação da coleta de tempo da execução da função `glDrawArrays` (responsável por realizar a renderização), foram plotados os gráficos – para cada *shader* – do tempo em nanosegundos *versus* a quantidade de polígonos. Porém, após feita as plotagens dos gráficos, foi possível perceber que as funções relacionadas a todos os *shaders* possuíam formas semelhantes (como é visto na Figura 23). Assim, chegou-se à conclusão de que era necessário coletar métricas específicas ao *vertex* e *fragment shaders*, a fim de poder investigar mais a fundo.

3.5.2 Medição do Número de Instruções por Segundo, Plotagem e Ajuste das Curvas

Para a coleta de medições relacionadas ao *vertex* e *fragment shaders*, utilizou-se a ferramenta *Adreno Profiler*. As métricas escolhidas foram a de número de instruções por segundo por vértice e número de instruções por segundo por fragmento. Estas métricas foram coletadas para cada número específico de polígonos, sendo os resultados exporta-

⁹ <http://www.khronos.org/registry/gles/>

dos no formato CSV. A fim de ajustar as curvas obtidas, e consequentemente estimar a complexidade algorítmica, utilizou-se o método dos mínimos quadrados (Seção 2.3.2) para funções lineares, quadráticas, cúbicas e exponenciais, calculando-se os respectivos erros, e podendo assim, determinar qual destas curvas melhor se aproximava das medições.

Para automatizar as etapas descritas anteriormente, foi feito um programa em Python que lê os arquivos CSV, faz a média das medições, plota os gráficos para o *vertex* e *fragment shaders* e realiza os ajustes das curvas, calculando os erros associados e determinando o menor entre eles. Ele também pode ler as medições a partir de um arquivo de extensão *txt* e realizar os ajustes das curvas para todo o processo de renderização (utilizando a medição do tempo em nanosegundos). Este programa (*script*) é executado por linha de comando, tendo como parâmetro o *shader* desejado e qual medição utilizada (se a relacionada a cada tipo de *shader* ou ao processo de renderização como um todo, como pode ser visto na Figura 24).

O programa foi estruturado de acordo com a Figura 25, em que a classe *ReadCSV* é responsável por ler os arquivos CSV e fazer a média das métricas tanto para o *vertex shader* como para o *fragment shader*. A classe *ReadTxt* é responsável por ler as medições de um arquivo de extensão *txt*, relacionado a todo o processo de renderização. Já a classe *PlotChart*, faz a plotagem dos gráficos do número de instruções por segundo por vértice/-fragmento (ou do tempo de renderização em nanosegundos) *versus* o número de polígonos. Além disso, ele também plota o gráfico original juntamente com o gráfico proveniente da aplicação do método dos mínimos quadrados. Por fim, o módulo *LeastSquares* realiza o ajuste dos mínimos quadrados para uma reta, uma exponencial e para polinômios de segundo e terceiro graus. Este módulo também calcula os erros associados a cada ajuste e indica o menor dos erros obtidos dentre todas as aproximações feitas.

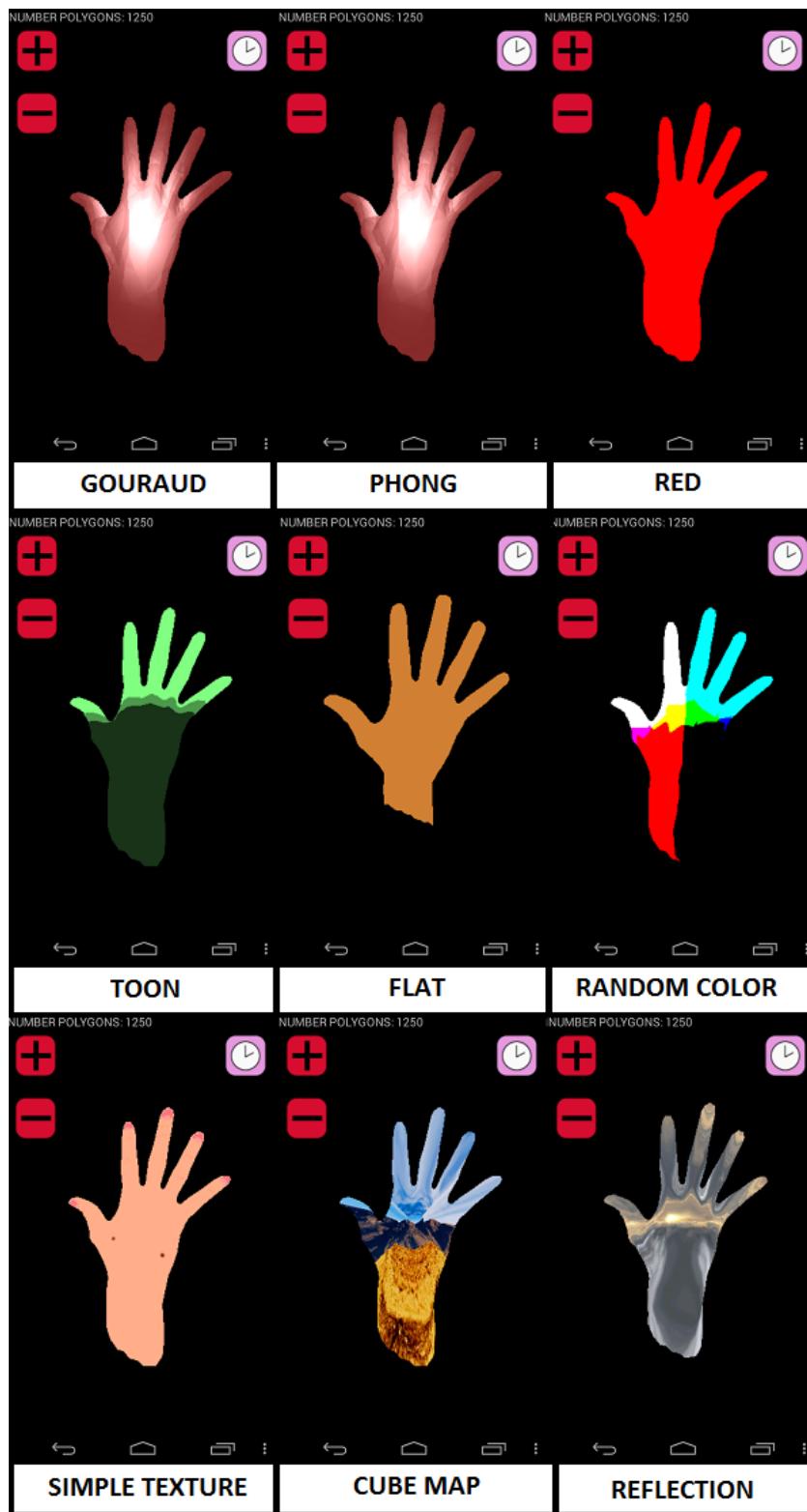


Figura 22 – *Shaders* Implementados

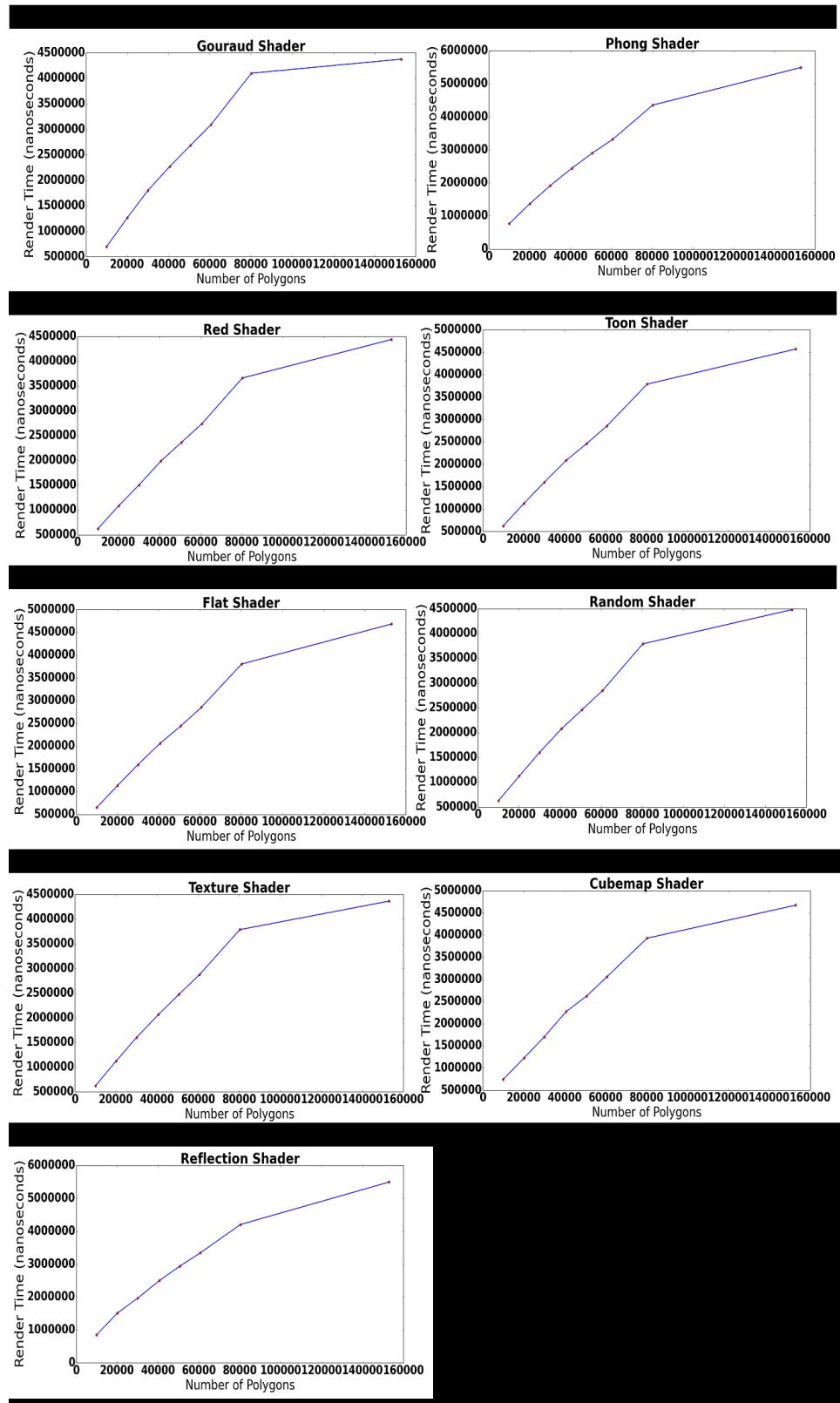


Figura 23 – Gráficos relacionados ao tempo de renderização em nanosegundos

```
C:\Users\Al_Campelo\Documents\UnB\Monografia\minquad>python shaderComplexity.py
phong render_time
C:\Users\Al_Campelo\Documents\UnB\Monografia\minquad>python shaderComplexity.py
phong number_instructions
```

Figura 24 – Linhas de comando para execução do programa

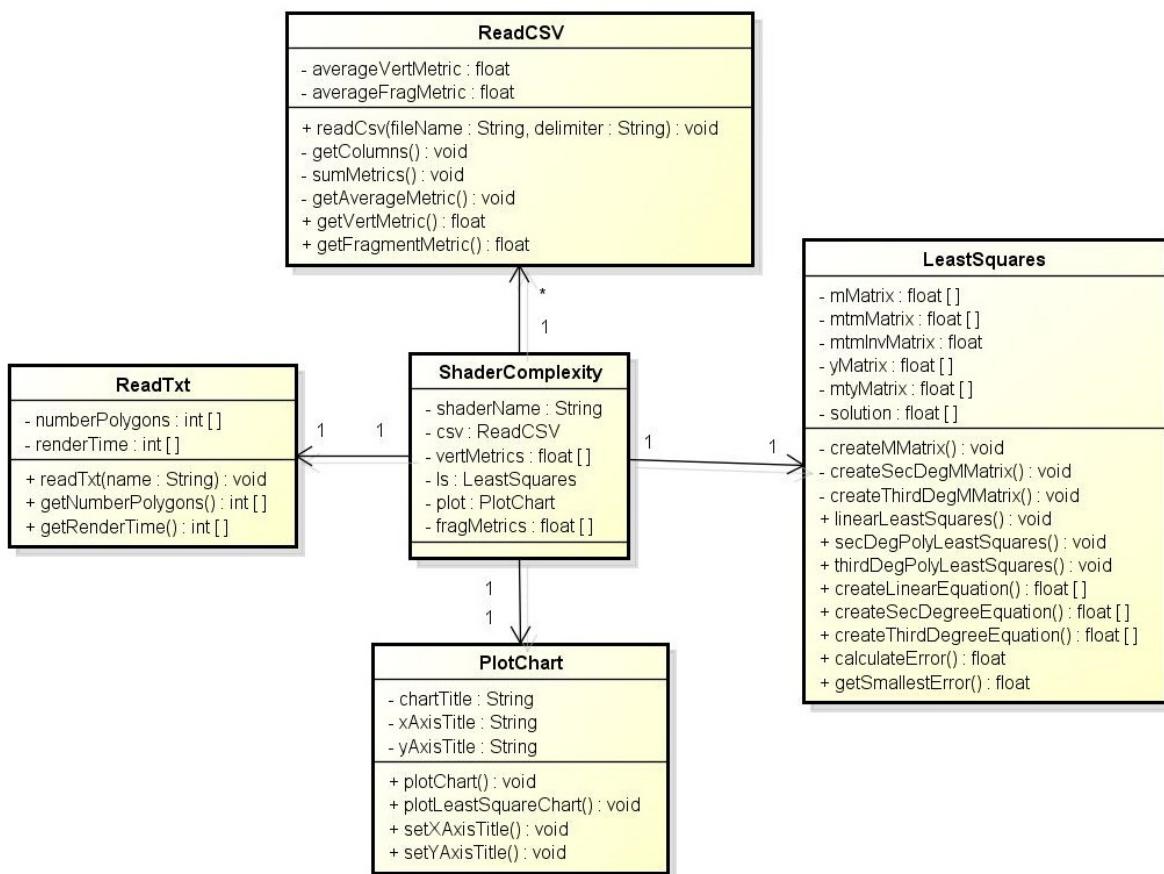


Figura 25 – Diagrama de Classes do *script* de automatização

4 Resultados

Para cada *shader* foram plotados os gráficos para as métricas relacionadas ao vértice e ao fragmento. Após as plotagens, percebeu-se que todos os gráficos de todos os *shaders* relacionados ao vértice deram uma função linear (diferindo na inclinação) e os relacionados ao fragmento deram uma curva de formato semelhante. A Figura 27 e a Figura 28 mostram os gráficos plotados, com relação ao *vertex* e *fragment shaders* de cada *shader* implementado, que demonstram a semelhança destas curvas.

Os ajustes em relação às curvas pré-definidas (linear, exponencial, segundo e terceiro graus) também foram calculados e plotados (Figura 30 referente ao *Reflection Shader*). Também foram determinados os menores erros associados, a fim de descobrir qual a curva melhor ajustava as medições do *fragment shader*. Pela análise do menor erro, calculado de acordo com a Seção 2.3.2, todos os *shaders* se aproximaram melhor de uma curva de segundo grau. Estes ajustes também foram feitos para as curvas obtidas do tempo do processo de renderização *versus* a quantidade de polígonos (que foi discutido na Seção 3.5.1). A análise do menor erro calculado indicou que, assim como o *fragment shader*, todas as curvas se aproximaram melhor de uma curva de segundo grau.

As equações calculadas para cada *shader* (relacionadas ao vértice e fragmento) são mostradas na Tabela 6. Embora as curvas sejam de mesma família, os seus coeficientes não são. Os *shaders* relativamente mais simples tem inclinação de reta menor, assim como o coeficiente do termo x^2 . Pela análise das equações, é possível perceber que o *vertex shader* de melhor desempenho é o do *Flat Shader*, que apenas determina as coordenadas x e y , já que o z é sempre zero. O de pior desempenho é o *Gouraud Shader*, que faz os cálculos de luz por vértice. Já o *fragment shader* de melhor desempenho é o do *Red Shader*, que apenas determina que a cor do fragmento seja vermelha. O de pior desempenho foi o do *Phong Shader*, que faz os cálculos de luz por fragmento.

Outra observação que pode ser feita é quanto aos *shaders Gouraud* e *Phong*, pois eles realizam o mesmo cálculo, porém o primeiro faz no *vertex shader* e o segundo, no *fragment shader*. Pela análise das equações, o desempenho relacionado ao *vertex shader* do *Gouraud* é pior que a do *Phong* e a relacionado ao *fragment shader* é melhor. Além disso, com as equações é possível estimar a quantidade de instruções por segundo por vértice ou por fragmento. Tomando como exemplo o *Toon Shader*, que sua equação para o *vertex shader* é $y = 10.17 \times 10^6 + 4673.96t$, o número de instruções por segundo por vértice estimado para 60000 polígonos é de 29.06×10^7 . Realizando a medição com a ferramenta *Adreno Profiler* foi possível perceber que este valor é próximo ao medido, que foi 28.49×10^7 .

As equações relacionadas a todo o processo de renderização também foram calculadas e podem ser vistas na Tabela 7. Assim como no caso anterior, elas são da mesma família mas possuem coeficientes diferentes. De acordo com estas equações, o *shader* de pior desempenho é o de Reflexão, que pela análise anterior, os *shaders* de vértice e fragmento eram uns dos de pior desempenho. Outro resultado relevante é quanto ao *Gouraud* e *Phong shaders*, em que na análise anterior, o primeiro possui o pior desempenho entre os *shaders* de vértice e o segundo, entre os *shaders* de fragmento. Mas o que possui o pior desempenho entre os dois, em relação ao processo de renderização como um todo, é o *Phong shader*. Este resultado é consistente, pois o *fragment shader*, pelo experimento realizado, possui complexidade algorítmica n^2 e o *vertex shader*, n , influenciando neste pior desempenho. Pelo experimento realizado, os *shaders* de melhores desempenhos são o *Flat*, *Toon* e *Red*. Além disso, utilizando as equações calculadas, a fim de estimar o tempo para 200000 polígonos, por exemplo, essa análise se confirma, como é mostrada na Tabela 8.

Assim, o processo utilizado neste trabalho de estimativa da complexidade algorítmica calculada de forma empírica, pode ser resumido na Figura 26. A etapa de Implementar *Shaders* pode ser feita por meio da utilização da base do projeto implementado, extendendo-se da classe *Shader* e implementando os métodos abstratos, como explicado na Seção 3.4. A etapa Realização das Medições é feita de forma manual, dependendo do *profiler* de GPU adequado para o *device* utilizado. E a etapa Plotar Gráficos, Ajustar Curvas e Obter Equações pode ser feita através do *script* criado para o ajuste das curvas.

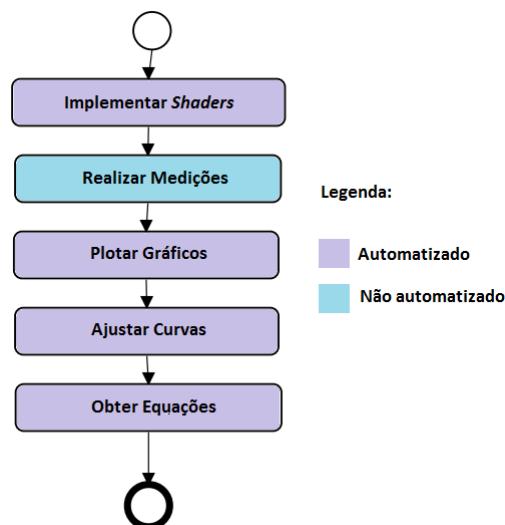


Figura 26 – Processo da Análise de Complexidade Algorítmica.

Nome	Instruções por Segundo por Vértice	Instruções por Segundo por Fragmento
<i>Gouraud</i>	$y = 40, 16 \times 10^6 + 7486, 43t$	$y = 19, 44 \times 10^8 + 187, 41t - 0, 0019t^2$
<i>Phong</i>	$y = 14, 95 \times 10^6 + 5211, 02t$	$y = 19, 87 \times 10^8 + 1034, 35t - 0, 0087t^2$
<i>Red</i>	$y = 8, 02 \times 10^6 + 4545, 69t$	$y = 19, 39 \times 10^8 + 53, 58t - 0, 00044t^2$
<i>Toon</i>	$y = 10, 17 \times 10^6 + 4673, 96t$	$y = 19, 44 \times 10^8 + 204, 84t - 0, 0017t^2$
<i>Flat</i>	$y = 7, 65 \times 10^6 + 3738, 61t$	$y = 19, 39 \times 10^8 + 57, 11t - 0, 00050t^2$
<i>Random Color</i>	$y = 20, 58 \times 10^6 + 5640, 13t$	$y = 19, 44 \times 10^8 + 170, 31t - 0, 0016t^2$
<i>Simple Texture</i>	$y = 8, 80 \times 10^6 + 4540, 32t$	$y = 19, 41 \times 10^8 + 112, 05t - 0, 0010t^2$
<i>CubeMap</i>	$y = 8, 67 \times 10^6 + 4540, 40t$	$y = 19, 43 \times 10^8 + 165, 99t - 0, 0014t^2$
<i>Reflection</i>	$y = 18, 03 \times 10^6 + 5470, 95t$	$y = 19, 59 \times 10^8 + 596, 54t - 0, 0051t^2$

Tabela 6 – Equações relacionadas ao *vertex shader*, *fragment shader*

Nome	Tempo do Processo de Renderização (ns)
<i>Gouraud</i>	$y = 36.43 \times 10^3 + 64.62t - 0.0002t^2$
<i>Phong</i>	$y = 62.66 \times 10^3 + 68.37t - 0.00021t^2$
<i>Red</i>	$y = -36.35 \times 10^3 + 58.8t - 0.00019t^2$
<i>Toon</i>	$y = -63.63 \times 10^3 + 62.91t - 0.00022t^2$
<i>Flat</i>	$y = -45.75 \times 10^3 + 62.31t - 0.00022t^2$
<i>Random Color</i>	$y = -43.67 \times 10^3 + 61.74t - 0.0002t^2$
<i>Simple Texture</i>	$y = -41.8 \times 10^3 + 61.72t - 0.0002t^2$
<i>CubeMap</i>	$y = 76.04 \times 10^3 + 61.65t - 0.00019t^2$
<i>Reflection</i>	$y = 251.35 \times 10^3 + 63.5t - 0.00019t^2$

Tabela 7 – Equações relacionadas ao tempo do processo de renderização

Nome	Tempo do Processo de Renderização (ns)
<i>Gouraud</i>	4960430
<i>Phong</i>	5336660
<i>Red</i>	4123650
<i>Toon</i>	3718370
<i>Flat</i>	3616250
<i>Random Color</i>	4304330
<i>Simple Texture</i>	4302200
<i>CubeMap</i>	4806040
<i>Reflection</i>	5351345

Tabela 8 – Exemplo de estimativa para 200000 polígonos

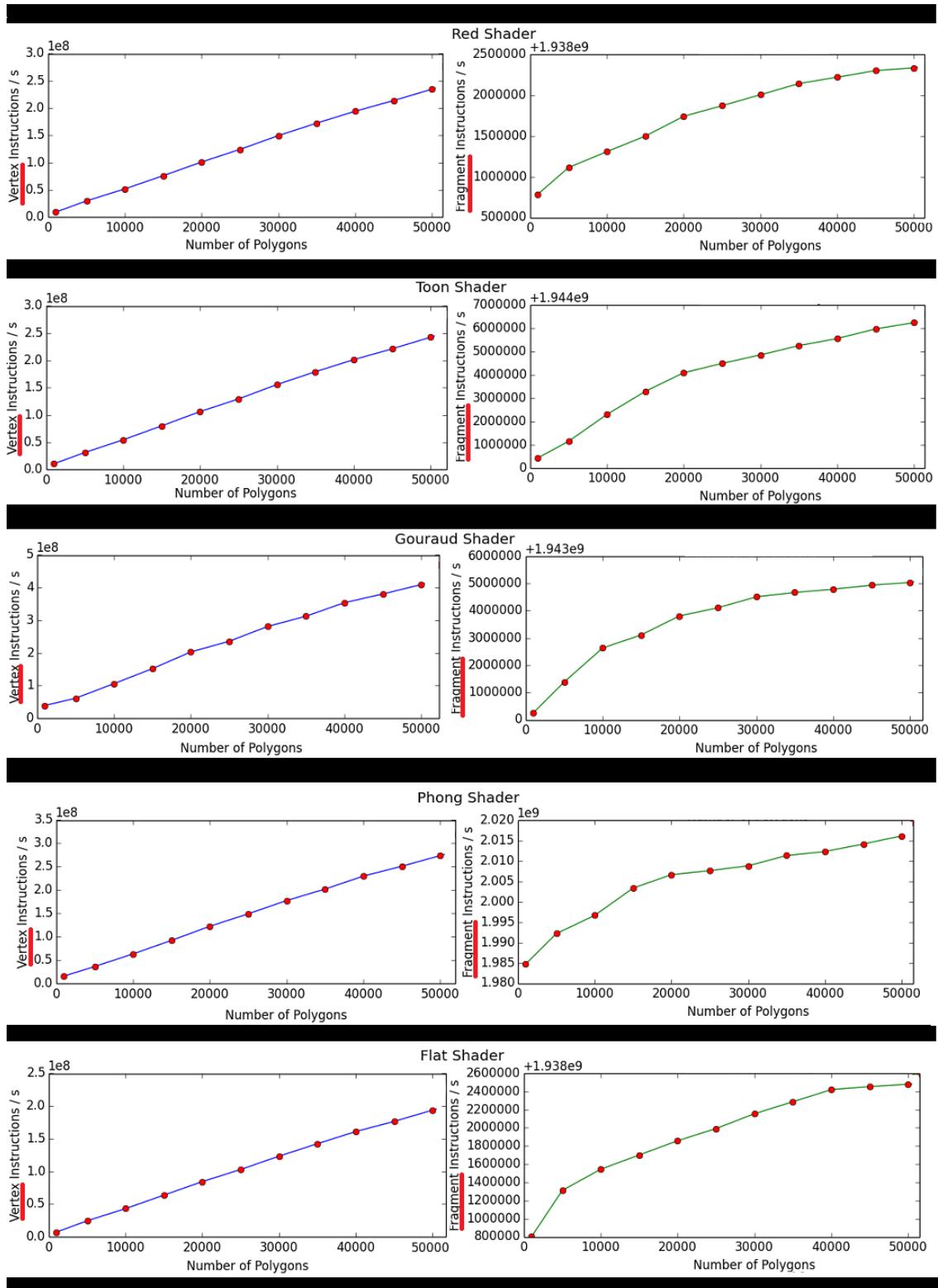


Figura 27 – Gráficos: *Red Shader*, *Toon shader*, *Gouraud Shader*, *Phong Shader* e *Flat Shader*

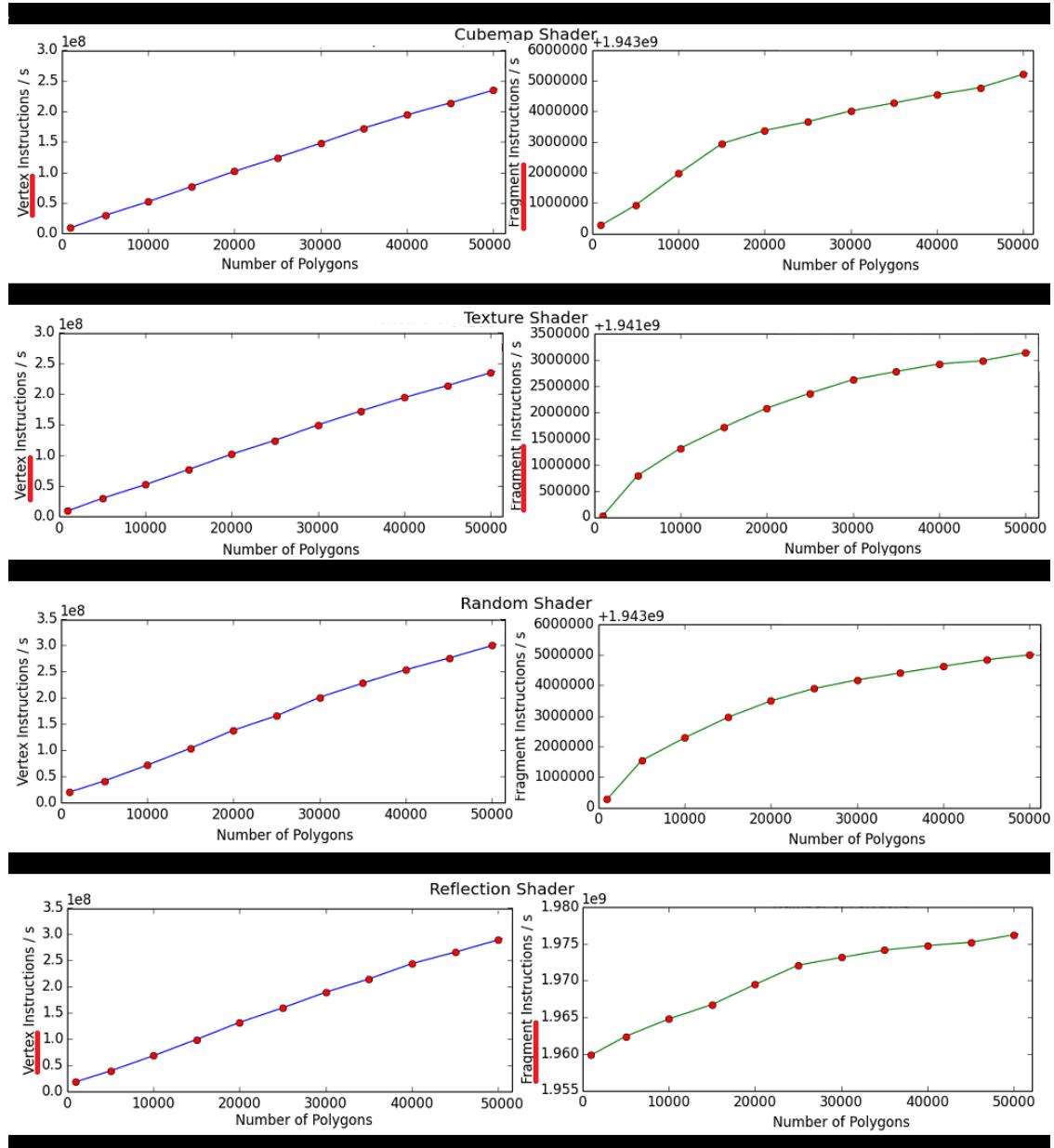


Figura 28 – Gráficos: *Cubemap Shader*, *Texture Shader*, *Random Shader*, *Reflection Shader*

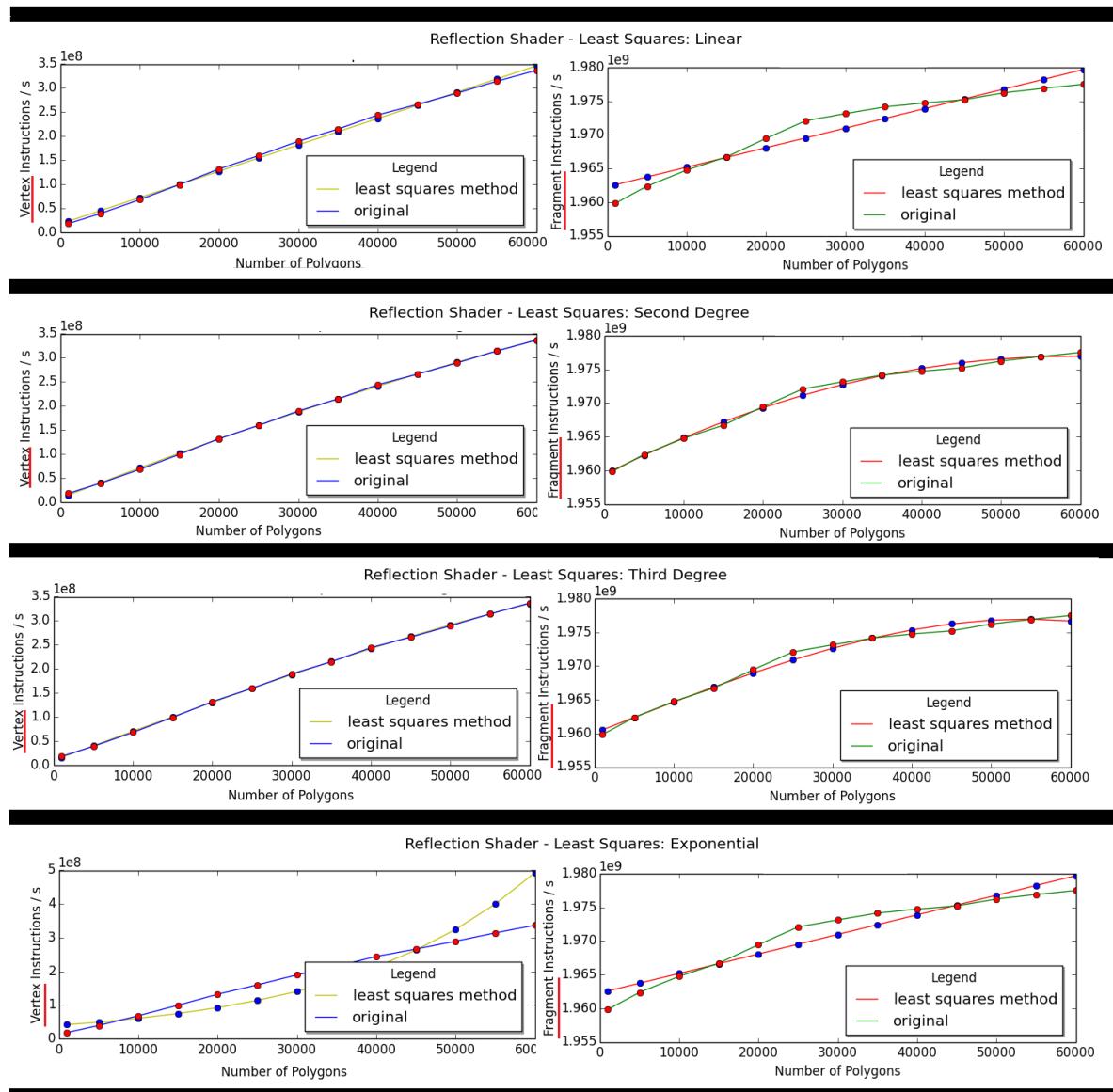


Figura 29 – Ajustes linear, segundo, terceiro graus e exponencial para cada tipo de *shader*

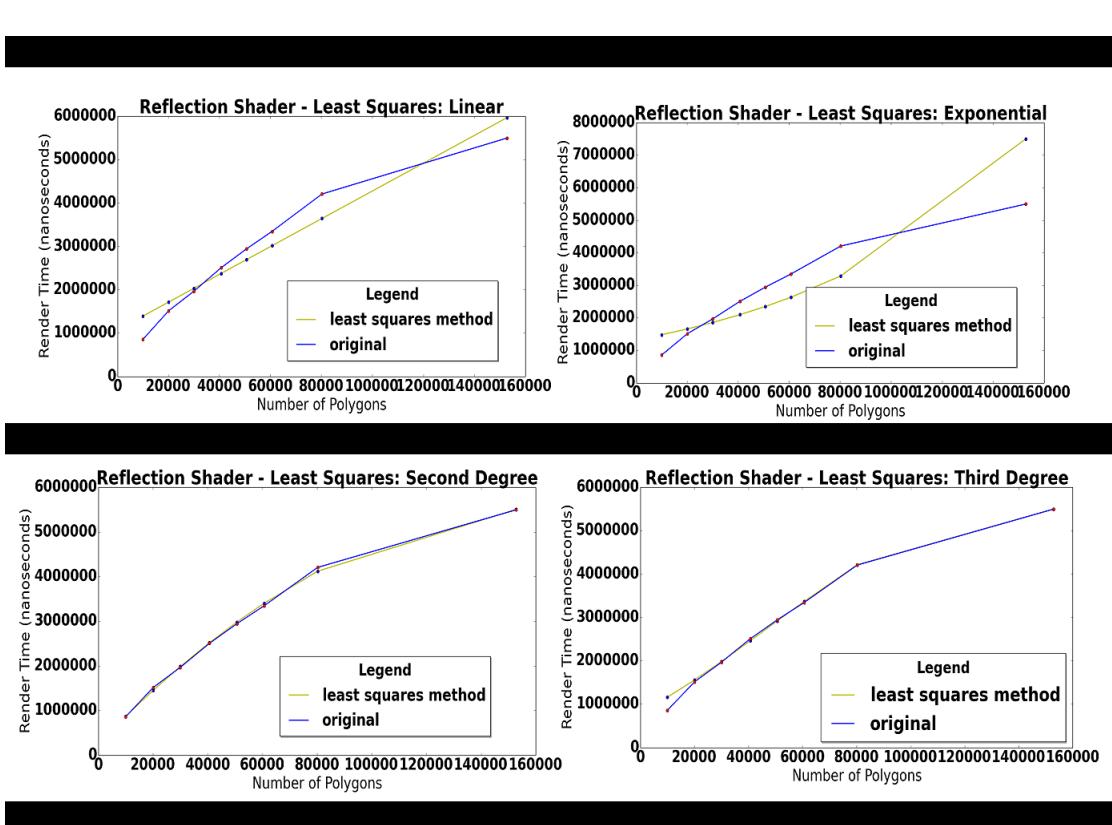


Figura 30 – Ajustes linear, segundo, terceiro graus e exponencial para processo de renderização

5 Conclusão

Por meio do trabalho realizado, foi possível concluir que o processo de renderização como um todo, calculado por meio do tempo de renderização da GPU, tendeu a apresentar como complexidade algorítmica um polinômio de segundo grau para qualquer *shader* (variando somente os coeficientes da função). Já o processo relacionado ao *vertex shader*, por meio dos experimentos realizados, foi possível perceber que a complexidade algorítmica se comportou linearmente e o relacionado ao *fragment shader* se aproximou de um polinômio de segundo grau, também independentemente do *shader* utilizado. Assim, todos os *shaders* possuem a mesma complexidade algorítmica, porém as equações de cada um possuem coeficientes diferentes, que podem determinar qual *shader* tem melhor ou pior desempenho.

Analizando a teoria do processo de renderização da *OpenGL* este resultado é consistente, pois o programa do *vertex shader* é utilizado para cada vértice (sendo de ordem linear). O do *fragment shader*, por sua vez, é de ordem do segundo grau, já que seu programa é usado para cada fragmento (sendo uma matriz). Além disso, ao analisar a documentação da GLSL¹, percebe-se que um fragmento (ou vértice) específico não tem conhecimento dos seus fragmentos (ou vértices) vizinhos, então não é possível que a complexidade cresça de um polinômio de segundo grau para um de terceiro, por exemplo. Porém este resultado não é tão óbvio, pois ao analisar o programa do *vertex shader* do Código 5.1, por exemplo, há somente uma atribuição, induzindo o programador a achar que a complexidade é constante. E por meio deste trabalho foi possível perceber que não é.

Código 5.1 – Exemplo de programa do *vertex shader*

```

1 uniform mat4 uMVPMatrix;
2 attribute vec4 aPosition;
3
4 void main() {
5     gl_Position = uMVPMatrix * aPosition;
6 }
```

Assim, como todos os *shaders* (do mesmo tipo) apresentam a mesma complexidade algorítmica, uma forma de comparar o desempenho entre eles, para um mesmo *device*, é através do processo realizado neste trabalho (explicado na Seção 4), que resulta no cálculo das funções de cada *shader*. Esta comparação pode ser feita por meio da análise destas

¹ <http://www.opengl.org/documentation/gsl/>

funções obtidas, comparando-se os seus coeficientes. Esta análise pode ser realizada com relação a todo o processo de renderização (utilizando a medida de tempo de renderização feita pela GPU) ou especificadamente ao *vertex shader* ou *fragment shader* – como neste trabalho, em que foram utilizados as medidas específicas de instruções por segundo por vértice/fragmento). Isto pode ser feito para comparar diferentes *shaders* ou para saber o quanto um *shader* foi otimizado (comparando-se o anterior e o atual).

Outra contribuição importante foi quanto à automatização da maior parte deste processo de análise da complexidade algorítmica, como a estrutura para aplicação dos *shaders*, média das medições, plotagem, ajuste das curvas e cálculo das funções. Assim, tal procedimento pode ser reproduzido de forma mais rápida e confiável.

Referências

- ARNAU, J.; PARCERISA, J.; XEKALAKIS, P. Teapot: A toolset for evaluating performance, power and image quality on mobile graphics systems. 27th Int. Conf. on Supercomputing, p. 37–46, 2013. Citado na página [17](#).
- BROTHALER, K. *OpenGL ES 2 for Android*: A quick-start guide. 1. ed. Dallas, Texas: The Pragmatic Bookshelf, 2013. Citado na página [69](#).
- DROZDEK, A. *Estrutura de Dados e Algoritmos em C++*. 2. ed. São Paulo, São Paulo: Cengage Learning, 2002. Citado 2 vezes nas páginas [30](#) e [31](#).
- EVANGELISTA, B.; SILVA, A. Creating photorealistic and non-photorealistic effects for games. Brazilian Symposium on Games and Digital Entertainment - SBGames 2007 Unisinos, 2007. Citado na página [28](#).
- GUHA, S. *Computer Graphics Through OpenGL*: From theory to experience. 1. ed. Boca Raton, Florida: CRC Press, 2011. Citado 3 vezes nas páginas [25](#), [26](#) e [27](#).
- JACKSON, W. *Learn Android App Development*. 1. ed. New York, New York: Apress, 2013. Citado 2 vezes nas páginas [24](#) e [37](#).
- LEITHOLD, L. *O Cálculo com Geometria Analítica*. 3. ed. São Paulo, São Paulo: Harbra, 1994. Citado na página [32](#).
- MOLLER, T. A.; HAINES, E.; HOFFMAN, N. *Real-Time Rendering*. 2. ed. Boca Raton, Florida: CRC Press, 2008. Citado na página [21](#).
- NADALUTTI, D.; CHITTARO, L.; BUTTUSSI, F. Rendering of x3d content on mobile devices with opengl es. Proc. Of 3D technologies for the World Wide Web, Seção Mobile devices, p. 19–26, 2006. Citado na página [17](#).
- QUALCOMM, D. N. *Mobile Gaming and Graphics Optimization (Adreno) Tools and Resources*. 2013. Disponível em: < <https://developer.qualcomm.com/mobile-development/mobile-technologies/gaming-graphics-optimization-adreno/tools-and-resources> >. Acessado em: 16 out. 2013. Citado na página [36](#).
- RORRES, A. *Álgebra Linear com Aplicações*. 8. ed. Porto Alegre, Rio Grande do Sul: Bookman, 2001. Citado na página [32](#).
- SANDBERG, R.; ROLLINS, M. *The Business of Android Apps Development*. 2. ed. New York, New York: Apress, 2013. Citado 2 vezes nas páginas [17](#) e [24](#).
- SEGEWICK, R. *Algorithms in C*. 1. ed. Westford, Massachusetts: Addison-Wesley, 1990. Citado na página [30](#).
- SHERROD, A. *Game Graphics Programming*. 1. ed. Boston, Massachusetts: Course Technology, 2011. Citado 2 vezes nas páginas [17](#) e [73](#).

SMITHWICK, M.; VERMA, M. *Pro OpenGL ES for Android*. 1. ed. New York, New York: Apress, 2012. Citado na página [25](#).

WRIGHT, R. S. et al. *OpenGL SuperBible: Comprehensive tutorial and reference*. 5. ed. Boston, Massachusetts: Pearson, 2008. Citado na página [21](#).

Anexos

ANEXO A – Processo de Renderização

O processo de geração de gráficos tridimensionais em computadores tem início com a criação de cenas. Uma cena é composta por objetos, que por sua vez são compostos por primitivas geométricas (como triângulos, quadrados, linhas, entre outros) que são constituídas de vértices, estabelecendo a geometria. Todos estes vértices seguem um processo similar de processamento para formarem uma imagem na tela. Este processo é mostrado na Figura 1 e as próximas seções detalham cada uma das etapas ilustradas.

A.1 Processamento dos Dados dos Vértices

A etapa de Processamento dos Dados dos Vértices é responsável por configurar os objetos utilizados para renderização com um *shader* específico, dependendo da técnica de renderização de modelos tridimensionais utilizada. Uma destas técnicas é a utilização de um *vertex array object*, que descreve o modelo tridimensional por meio de uma lista de vértices e uma lista de índices. Na Figura 31, tem-se dois triângulos e quatro vértices definidos (dois vértices são compartilhados). Assim, pode-se definir um vetor com os vértices $[v_0, v_1, v_2, v_3]$ e um vetor de índices $[0, 3, 1, 0, 2, 3]$, que determina a ordem em que os vértices devem ser renderizados.

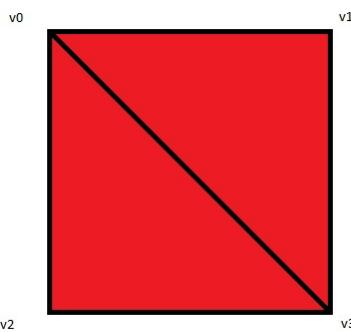


Figura 31 – Vértices do quadrado constituído de dois triângulos

Outra técnica é a utilização de um *vertex object buffer*, em que a ideia é parecida com a da técnica anterior, porém, de acordo com (BROTHALER, 2013), o *driver* gráfico pode optar por colocar o *buffer* contendo os vértices e índices diretamente na memória da GPU, melhorando o desempenho para objetos que não são modificados com muita frequência.

A.2 Processamento dos Vértices

É nesta etapa que modela-se parte dos efeitos visuais (a outra parte é feita durante o processamento dos fragmentos). Estes efeitos incluem as características dos materiais atribuídos aos objetos, como também os efeitos da luz, sendo que cada efeito pode ser modelado de diferentes formas, como representações de descrições físicas. Muitos dados são armazenados em cada vértice, como a sua localização e o vetor normal associado (que indica a orientação do vértice no espaço), por exemplo. O *vertex shader* é aplicado nesta etapa.

Além disso, as transformadas são aplicadas nas coordenadas do objeto, de forma que ele possa ser posicionado, orientado e tenha um tamanho determinado. Após o ajuste das coordenadas, é dito que o objeto está localizado no espaço do mundo e, em seguida, é aplicada a transformação de visualização, que tem como objetivo estabelecer a câmera. A etapa de projeção é responsável por transformar o volume de visualização aplicando métodos de projeção, como a perspectiva e a ortográfica (também chamada de paralela). A projeção ortográfica resulta em uma caixa retangular, em que linhas paralelas permanecem paralelas após a transformação. Na perspectiva, quanto mais longe um objeto se encontra, menor ele aparecerá após a projeção: linhas paralelas tendem a convergir no horizonte. Ela resulta em um tronco de pirâmide com base retangular.

A.3 Pós-Processamento dos Vértices

Durante a etapa de Pós-Processamento dos Vértices, os dados da etapa anterior são guardados em *buffers*. Além disso, as primitivas geradas pelas etapas anteriores poderão ser recortadas caso estejam fora do volume de visão. Ou seja, somente as primitivas gráficas que se encontram dentro do volume de visualização serão renderizadas. Assim, o recorte (chamado *clipping*) é responsável por não passar adiante as primitivas que se encontram fora da visualização. Primitivas que estão parcialmente dentro são recortadas, ou seja, o vértice que está de fora não é renderizado e é substituído por um novo vértice (dentro do volume de visualização). A Figura 32 ilustra esta ideia.

A.4 Montagem das Primitivas

A Montagem das Primitivas é o estágio em que as primitivas a serem renderizadas são enviadas. Estas primitivas são discretizadas em pontos, linhas e triângulos, em que algumas delas podem ser descartadas, baseando-se nas faces aparentes (procedimento conhecido como *Face Culling*).

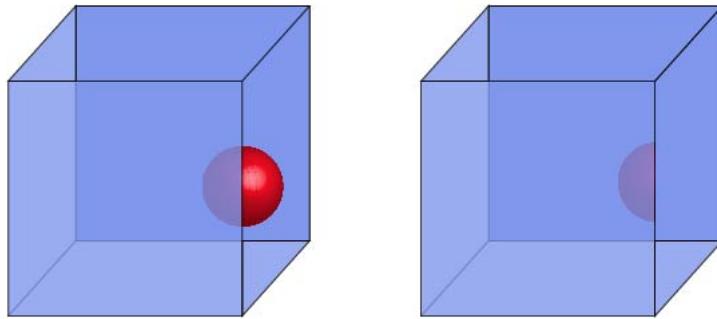


Figura 32 – Antes do recorte (cubo de visualização esquerdo) e depois do recorte (cubo de visualização direito)

A.5 Conversão e Interpolação dos Parâmetros das Primitivas

Nesta etapa que ocorre o procedimento conhecido como rasterização. Nela, cada primitiva é transformada em fragmentos, cuja entrada é formada pelas primitivas recortadas e as coordenadas ainda tridimensionais. Assim, esta etapa tem como finalidade mapear as coordenadas tridimensionais em coordenadas de tela. Para isto, o centro de um *pixel* (*picture element*) é igual à coordenada 0,5. Então, *pixels* de [0; 9] equivalem à cobertura das coordenadas de [0,0; 10,0]. E os valores dos *pixels* crescem da esquerda para a direita e de cima para baixo. Também é feita a configuração dos triângulos, em que dados são computados para as superfícies dos triângulos. Eles serão utilizados para a conversão dos dados vindos do processo de geometria (coordenadas e informações provenientes do *vertex shader*) em *pixels* na tela e também para o processo de interpolação. Assim, é checado se cada um dos *pixels* está dentro de um triângulo ou não. Para cada *pixel* que sobrepõe um triângulo, um fragmento é gerado, conforme mostrado na Figura 33. Cada fragmento tem informações sobre sua localização na tela, no triângulo e sua profundidade, e as propriedades dos fragmentos dos triângulos são geradas usando dados interpolados entre os três vértices do triângulo.

A.6 Processamento dos Fragmentos

As computações por *pixel* são calculadas durante o *Fragment Shading*, em que o resultado consiste em uma ou mais cores a serem passadas para o próximo estágio. Muitas técnicas podem ser aplicadas durante esta etapa, em que uma delas é a de texturização (que aplica no fragmento do objeto parte de uma imagem).

A.7 Processamento das Amostras

A informação relacionada com cada *pixel* é armazenada no *color buffer*, que é um *array* de cores. Assim, a última etapa é a de Processamento das Amostras, que realiza

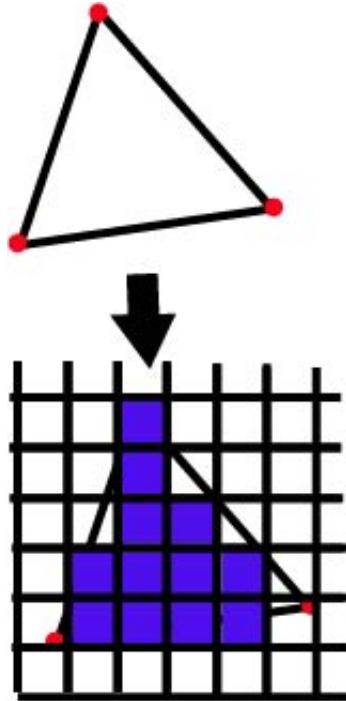


Figura 33 – Travessia de triângulos: fragmentos sendo gerados

diversos testes nos fragmentos gerados na etapa anterior. Ela é responsável, por exemplo, pelos testes de profundidade, em que o *color buffer* deve conter as cores das primitivas da cena que são visíveis do ponto de vista da câmera. Isto é feito através do *Z-buffer* (também chamado de *buffer* de profundidade), que para cada *pixel* armazena a coordenada *z* a partir da câmera até a primitiva mais próxima. Então, a coordenada *z* de uma primitiva que está sendo computada é comparada com o valor do *Z-buffer* para o mesmo *pixel*. Se o valor for menor, quer dizer que a primitiva está mais próxima da câmera do que o valor da anterior, e assim, o valor do *Z-buffer* é atualizado para o atual. Se o valor corrente for maior, então o valor do *Z-buffer* não é modificado. Outros testes realizados são o de *blending*, em que combina-se as cores do fragmento com a do *buffer* e os de descarte de fragmentos como o *scissor test* e *stencil test*.

ANEXO B – Representação de Objetos Tridimensionais: Formato *obj*

Em uma cena, os modelos tridimensionais podem variar muito mais do que formas básicas como uma esfera e um torus, por exemplo. Assim, o formato *obj* foi criado pela empresa *Wavefront* e é um arquivo para leitura de objetos tridimensionais, a fim de carregar geometrias mais complexas. Segundo ([SHERRON, 2011](#)), neste arquivo cada linha contém informações a respeito do modelo, começando com uma palavra-chave, seguida da informação. A Tabela 9 mostra as principais palavras-chave utilizadas.

Palavra-chave	Significado
usemtl	Indica se está utilizando material
mtlib	Nome do material
v	Coordenadas x, y e z do vértice
vn	Coordenadas da normal
vt	Coordenadas da textura
f	Face do polígono

Tabela 9 – Palavras-chave do formato *obj*

A face do polígono (f) possui três índices que indicam os vértices do triângulo. Assim, cada vértice possui um índice (que depende de quando ele foi declarado), começando a partir de um. O código B.1 mostra o exemplo de um arquivo *obj* para a leitura de um cubo.

Código B.1 – Representação do formato *obj*

```

1 #Formato OBJ - Cubo
2
3 #8 vertices
4 v 2.0 -2.0 -2.0
5 v 2.0 -2.0 2.0
6 v -2.0 -2.0 2.0
7 v -2.0 -2.0 -2.0
8 v 2.0 2.0 -2.0
9 v 2.0 2.0 2.0
10 v -2.0 2.0 2.0
11 v -2.0 2.0 -2.0
12
13 #4 coordenadas de textura
14 vt 0.0 0.0

```

```
15 vt 1.0 0.0
16 vt 1.0 1.0
17 vt 0.0 1.0
18
19 #8 vetores normais
20 vn 0.578387 0.575213 -0.578387
21 vn 0.576281 -0.579455 -0.576281
22 vn -0.576250 -0.576281 -0.579455
23 vn -0.578387 0.578387 -0.575213
24 vn -0.577349 -0.577349 0.577349
25 vn -0.577349 0.577349 0.577349
26 vn 0.579455 -0.576281 0.576281
27 vn 0.575213 0.578387 0.578387
28
29 #6 faces - 12 triangulos
30 f 5/1/1 1/2/2 4/3/3
31 f 5/1/1 4/3/3 8/4/4
32 f 3/1/5 7/2/6 8/3/4
33 f 3/1/5 8/3/4 4/4/3
34 f 2/1/7 6/2/8 3/4/5
35 f 6/2/8 7/3/6 3/4/5
36 f 1/1/2 5/2/1 2/4/7
37 f 5/2/1 6/3/8 2/4/7
38 f 5/1/1 8/2/4 6/4/8
39 f 8/2/4 7/3/6 6/4/8
40 f 1/1/2 2/2/7 3/3/5
41 f 1/1/2 3/3/5 4/4/3
```

Então, a partir da leitura do arquivo *obj*, é possível ler cada linha e armazenar em estruturas de dados as informações que serão passadas para renderizar o modelo tridimensional, como vértices e índices, e utilizar um dos métodos apresentados anteriormente para renderizar a cena.