

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

Aproximação experimental da complexidade assintótica de *shaders* para dispositivos móveis utilizando *OpenGL ES*

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

Brasília, DF
2014



Alex de Souza Campelo Lima

**Aproximação experimental da complexidade assintótica
de *shaders* para dispositivos móveis utilizando *OpenGL ES***

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*.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF

2014

Alex de Souza Campelo Lima

Aproximação experimental da complexidade assintótica de *shaders* para dispositivos móveis utilizando *OpenGL ES*

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, 25 de junho de 2014:

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 tem crescido e constata-se a importância dos efeitos visuais em jogos, bem como a limitação atual de desempenho dos processadores gráficos destas plataformas. Assim, a proposta do trabalho se baseia na implementação e aproximação experimental da complexidade assintótica de *shaders* (programas responsáveis pelos efeitos visuais) para a plataforma *Android* e *iOS*. Suas complexidades assintóticas 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, permitindo determinar qual curva mais se aproxima da função original.

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

Abstract

The usage of mobile devices 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 and iOS platform. Their asymptotic 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, being able to estimate which curve has better approximation to the original function.

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

Lista de ilustrações

Figura 1 – Processo de renderização da <i>OpenGL</i>	21
Figura 2 – Utilização dos <i>shaders</i>	23
Figura 3 – Comparação entre as técnicas de <i>shading</i>	27
Figura 4 – Exemplo de escolha dos tons	28
Figura 5 – <i>Environment Map</i>	28
Figura 6 – Comparação da Complexidade Assintótica	30
Figura 7 – Ambiente de desenvolvimento <i>Eclipse</i>	36
Figura 8 – Novo projeto de um aplicativo na ferramenta <i>Xcode</i>	37
Figura 9 – Ferramenta de modelagem tridimensional	38
Figura 10 – Analisador de <i>OpenGL ES</i>	38
Figura 11 – Ferramenta <i>Adreno Profiler</i> : analisador de <i>shaders</i>	39
Figura 12 – Ferramenta <i>Adreno Profiler</i> : visualização de métrica quadros por segundo	40
Figura 13 – Diagrama de Classe da Implementação em <i>Android</i>	41
Figura 14 – Detalhamento das classes <i>Shader Activity</i> , <i>Splash Activity</i> e <i>Resources</i> .	41
Figura 15 – Tela da <i>Splash Activity</i>	42
Figura 16 – Tela da <i>Shader Activity</i>	43
Figura 17 – Detalhamento das classes <i>3DObject</i> e <i>Texture</i>	43
Figura 18 – Ordem das coordenadas de posição, normal e textura para um vértice .	43
Figura 19 – Técnica de mapeamento de textura utilizada para cada modelo 3D .	44
Figura 20 – Textura gerada a partir da técnica de mapeamento	45
Figura 21 – Detalhamento das classes <i>Timer</i> e <i>NativeLib</i>	45
Figura 22 – Detalhamento da classe <i>Renderer</i>	46
Figura 23 – Detalhamento da classe <i>Shader</i>	47
Figura 24 – <i>Shaders</i> Implementados	56
Figura 25 – Diagrama de Classes da Implementação em <i>iOS</i>	57
Figura 26 – <i>Gouraud Shader</i> na plataforma <i>iOS</i>	58
Figura 27 – Diagrama de Classes do <i>script</i> de automatização	60
Figura 28 – Gráficos: <i>Red Shader</i> , <i>Toon shader</i> , <i>Gouraud Shader</i> , <i>Phong Shader</i> e <i>Flat Shader</i> para o <i>Nexus 4</i>	63
Figura 29 – Gráficos: <i>Cubemap Shader</i> , <i>Texture Shader</i> , <i>Random Shader</i> , <i>Reflection</i> <i>Shader</i> para o <i>Nexus 4</i>	64
Figura 30 – Gráficos relacionados ao tempo de renderização em nanosegundos para o <i>Nexus 4</i>	65
Figura 31 – Comparações entre as curvas dos <i>shaders</i> : todo processo de renderização	66
Figura 32 – Comparações entre as curvas dos <i>shaders</i> : <i>vertex</i> e <i>fragment shaders</i> .	66

Figura 33 – Ajustes linear, segundo, terceiro graus e exponencial de cada tipo de <i>shader</i> para o <i>Nexus 4</i>	67
Figura 34 – Ajustes linear, segundo, terceiro graus e exponencial do processo de renderização para o <i>Nexus 4</i>	68
Figura 35 – Gráficos do <i>Gouraud Shader</i> para o <i>HTC One</i>	68
Figura 36 – Ajustes linear, segundo, terceiro graus e exponencial de cada tipo de <i>shader</i> para o <i>HTC One</i>	69
Figura 37 – Comparação entre as curvas dos <i>shaders</i> : <i>Nexus 4</i> e <i>HTC One</i>	70
Figura 38 – Gráficos relacionados ao tempo de renderização em nanosegundos para os dispositivos <i>iOS</i>	71
Figura 39 – Ajustes linear, segundo, terceiro graus e exponencial do processo de renderização para dispositivo <i>iOS</i>	71
Figura 40 – Comparação entre as curvas dos <i>shaders</i> : <i>Nexus 4</i> , <i>iPhone 5s</i> e <i>iPad Air</i>	72
Figura 41 – Processo da Análise de Complexidade Algorítmica.	78
Figura 42 – Vértices do quadrado constituído de dois triângulos	85
Figura 43 – Antes do recorte (cubo de visualização esquerdo) e depois do recorte (cubo de visualização direito)	87
Figura 44 – Travessia de triângulos: fragmentos sendo gerados	88

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 – Versões da plataforma <i>Android</i>	25
Tabela 6 – Valores mais comuns de complexidade assintótica	30
Tabela 7 – Dispositivos móveis utilizados	35
Tabela 8 – Equações relacionadas ao <i>vertex shader</i> e <i>fragment shader</i>	73
Tabela 9 – Equações relacionadas ao tempo do processo de renderização	74
Tabela 10 – Exemplo de estimativa para 170.000 polígonos	74
Tabela 11 – Equações do processo de renderização do <i>Gouraud Shader</i>	74
Tabela 12 – Equações do <i>vertex</i> e <i>fragment shaders</i> para o <i>Gouraud Shader</i>	75
Tabela 13 – Polinômios de segundo grau relacionados ao <i>fragment shader</i>	76
Tabela 14 – Polinômios de segundo grau relacionados ao tempo do processo de ren- derização	76
Tabela 15 – Equações do processo de renderização do <i>Gouraud Shader</i>	76
Tabela 16 – Equações do <i>vertex</i> e <i>fragment shaders</i> para o <i>Gouraud Shader</i>	76
Tabela 17 – Estimativa de <i>FPS</i>	77
Tabela 18 – Palavras-chave do formato <i>obj</i>	89

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>
GLSL	<i>OpenGL Shading Language</i>
CPU	<i>Central Processing Unit</i>
NDK	<i>Native Development Kit</i>
JNI	<i>Java Native Interface</i>
FPS	<i>Frames Per Second</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	Plataforma <i>iOS</i>	24
2.1.3.3	<i>OpenGL ES</i>	25
2.2	Fundamentação Matemática e Física para Implementação de <i>Shaders</i>	25
2.2.1	Renderização <i>Flat</i> , <i>Gouraud</i> e <i>Phong</i>	26
2.2.2	Renderização de Efeito <i>Cartoon</i>	27
2.2.3	Renderização de Efeito de Reflexão	28
2.3	Complexidade Assintótica	29
2.3.1	Abordagem Teórica	29
2.3.1.1	Notação <i>Big-O</i>	29
2.3.2	Abordagem Experimental	31
2.3.2.1	Ajuste Linear	31
2.3.2.2	Ajuste Exponencial	31
2.3.2.3	Ajuste para Polinômio de Segundo e Terceiro Grau	32
2.3.2.4	Cálculo dos Erros dos Ajustes	32
3	Desenvolvimento	35
3.1	Levantamento Bibliográfico	35
3.2	Equipamentos Utilizados	35
3.3	Configuração do Ambiente e Ferramentas	36
3.3.1	Plataforma <i>Android</i>	36
3.3.2	Plataforma <i>iOS</i>	36
3.3.3	Modelagem dos Objetos Tridimensionais	37
3.3.4	Coleta de Medições	37
3.3.5	Automatização de Cálculo e Plotagem	40

3.3.6	Controle de Versionamento	40
3.4	Implementação na Plataforma <i>Android</i>	40
3.4.1	Tela de <i>Front-end</i>	41
3.4.2	Objeto Tridimensional	43
3.4.3	Cálculo do Tempo de Renderização	44
3.4.4	Renderização	45
3.4.5	<i>Shaders</i>	46
3.4.5.1	<i>Phong Shader</i>	47
3.4.5.2	<i>Gouraud Shader</i>	48
3.4.5.3	<i>Red Shader</i>	50
3.4.5.4	<i>Toon Shader</i>	50
3.4.5.5	<i>Flat Shader</i>	51
3.4.5.6	<i>Random Color Shader</i>	52
3.4.5.7	<i>Simple Texture Shader</i>	53
3.4.5.8	<i>CubeMap Shader</i>	54
3.4.5.9	<i>Reflection Shader</i>	54
3.5	Implementação na Plataforma <i>iOS</i>	57
3.6	Estimativa Experimental da Complexidade Algorítmica	58
3.6.1	Medição do Processo de Renderização	58
3.6.2	Medição do <i>Vertex</i> e <i>Fragment Shaders</i>	59
3.6.3	Plotagem	59
3.6.4	Automatização dos Ajustes das Curvas	59
4	Resultados	61
4.1	Dispositivos <i>Android</i>	61
4.2	Dispositivos <i>iOS</i>	71
4.3	Análise das Equações Obtidas	72
4.3.1	Comparação das Equações dos Dispositivos	73
4.3.2	Considerações Finais das Equações	75
4.4	Estimativas em Ambientes de Produção	77
4.5	Processo Experimental da Estimação da Complexidade Assintótica	78
5	Conclusão	79
Referências		81
Anexos		83
ANEXO A Processo de Renderização		85
A.1	Processamento dos Dados dos Vértices	85

A.2	Processamento dos Vértices	86
A.3	Pós-processamento dos Vértices	86
A.4	Montagem das Primitivas	87
A.5	Conversão e Interpolação dos Parâmetros das Primitivas	87
A.6	Processamento dos Fragmentos	88
A.7	Processamento das Amostras	88
ANEXO B	Representação de Objetos Tridimensionais: Formato <i>obj</i>	89

1 Introdução

1.1 Contextualização e Justificativa

Conforme apontado em (SHERROD, 2011), em jogos os gráficos 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 há de ser atingido.

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 efeitos visuais mais complexos se tornem mais difíceis ainda.

O recente crescimento do desempenho de dispositivos móveis tornou-os capazes de suportar aplicações cada vez 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, segundo o CEO da *Apple*, Tim Cook¹, mais de 800 milhões de aparelhos utilizando a plataforma *iOS* já foram vendidos e a ativação diária de aparelhos na plataforma *Android*, de acordo com (SANDBERG; ROLLINS, 2013), é de aproximadamente 1,5 milhões, sendo uns dos sistemas operacionais para dispositivos móveis mais utilizados.

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

Assim, é possível analisar o desempenho do processo de renderização feito pela GPU – no qual diferentes *shaders* (responsáveis pela criação dos efeitos visuais) são aplicados – por meio da complexidade assintótica. Além disso, a análise do desempenho dos *shaders* é uma área pouco explorada, tornando o tema abordado neste trabalho relevante.

¹ <http://www.theverge.com/2014/6/2/5772344/apple-wwdc-2014-stats-update>

Então, o tema consiste no desenvolvimento de *shaders* aplicados em objetos tridimensionais – com número de polígonos variável – que permitam a coleta de medições relacionadas ao desempenho de um dispositivo em relação à renderização feita pela GPU. Desta forma, é possível variar a quantidade de polígonos de um objeto e traçar um gráfico da quantidade de polígonos *versus* métrica de desempenho, utilizando um determinado *shader*. E assim, é possível identificar qual curva melhor aproxima e, consequentemente, determinar a complexidade assintótica do *shader* em um determinado dispositivo.

1.2 Objetivos Gerais

O objetivo geral deste trabalho é a análise da complexidade assintótica de *shaders* para diferentes dispositivos móveis, tanto para todo o processo de renderização quanto para somente parte dele (*vertex* e *fragment shaders*).

1.3 Objetivos Específicos

Os objetivos específicos do trabalho são:

- Configurar os ambientes de desenvolvimento para a plataforma *Android* e *iOS*;
- Implementar os *shaders* para a plataforma *Android* e *iOS*;
- Procurar 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;
- Estimar a complexidade assintótica de *shaders*, de acordo com as 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 assintótica, método dos mínimos quadrados, entre outros.

No desenvolvimento, os passos seguidos ao longo do trabalho são descritos, enfatizando como foi feita a configuração do ambiente, quais equipamentos foram utilizados,

como foram feitas as implementações dos *shaders* e a automatização dos cálculos e plotagem dos gráficos.

Nos resultados são descritos os resultados obtidos através da análise da complexidade assintótica dos *shaders* implementados, seguido das conclusões do trabalho realizado, onde são apresentadas as contribuições obtidas.

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 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 mostrado na Figura 1, em que uma aplicação pode substituir as funcionalidades fixas aos vértices e aos fragmentos. O Anexo A descreve as etapas do processo ilustrado.

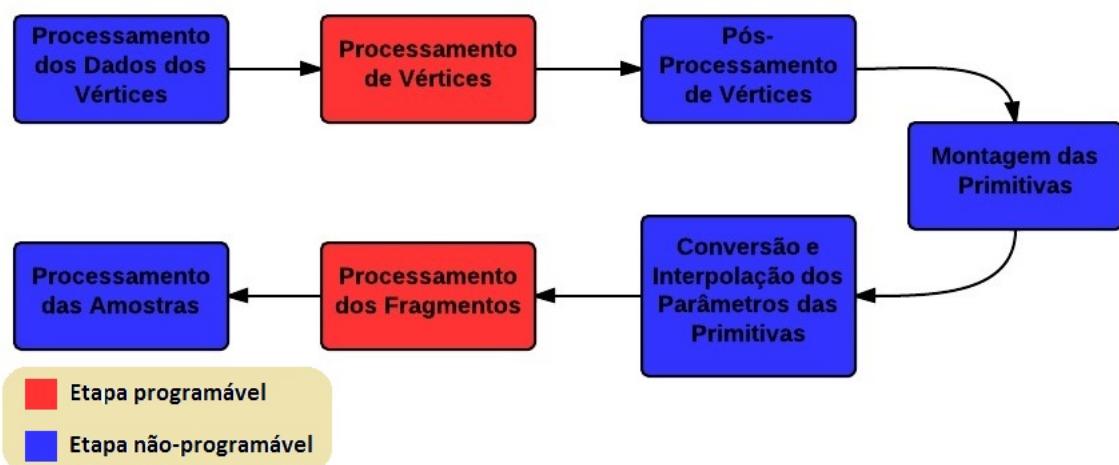


Figura 1 – Processo de renderização da *OpenGL*

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, a partir das coordenadas de posição, normal e textura, por exemplo. Ele altera a etapa de pro-

cessamento de vértices e deve, ao menos, definir as coordenadas de posição. O *fragment shader* opera na etapa de processamento dos fragmentos, e 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.

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

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*

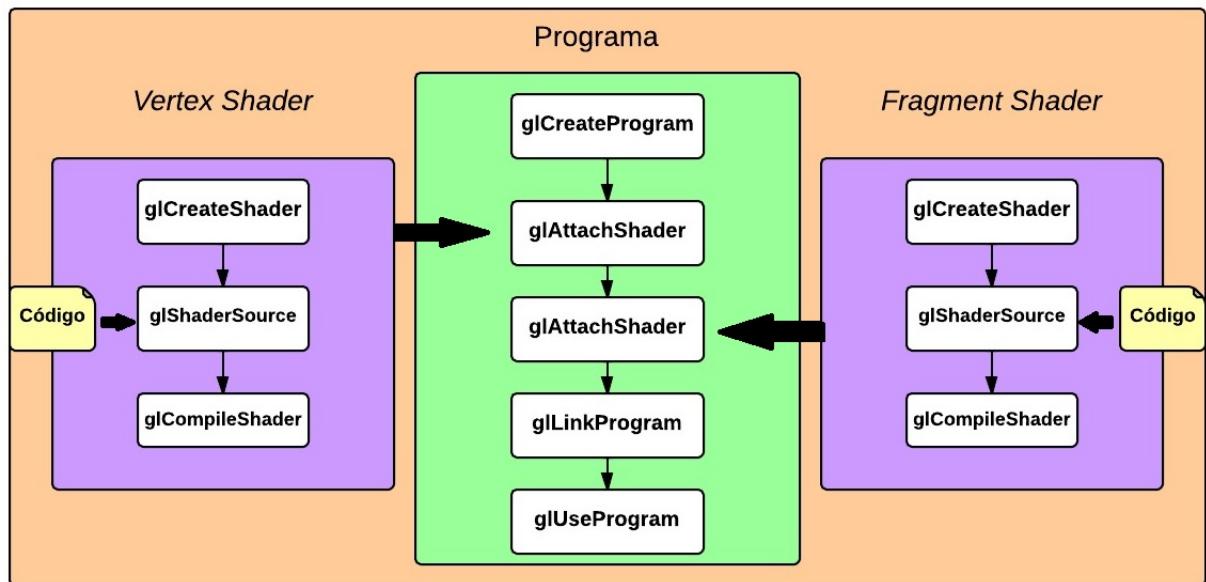
Para cada *shader* – relativos ao vértice e ao fragmento – é necessário escrever o código que será compilado e feito o *link*, gerando um programa final que será utilizado pela *OpenGL*. Ou seja: é feita a compilação e o *link* pela aplicacão de cada um destesshaders. Caso os *shaders* utilizem variáveis passadas pela *OpenGL*, é necessário primeiramente

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

Tabela 3 – GLSL: qualificadores

encontrar a localização desta variável e depois setá-la. Se a variável for do tipo *uniform*, por exemplo, adquire-se a localização por meio da função `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. Nas próximas seções são mostradas as plataformas *Android* e *iOS*, que foram utilizadas neste trabalho, e a biblioteca gráfica *OpenGL ES*, que é uma variação da *OpenGL* para dispositivos móveis, usada na 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 une 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. 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 chegaram a 1,5 milhões diários.

O *Android* também possui um mercado centralizado acessível por qualquer aparelho (*tablet* ou *smartphone*) chamado *Google Play*¹, facilitando a publicação e aquisição de aplicativos. Ele também possui diferentes versões, sendo elas mostradas na Tabela 4. 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*

2.1.3.2 Plataforma *iOS*

A plataforma *iOS* é uma plataforma móvel que foi lançada em 2007 e é distribuída exclusivamente para *hardware* fabricado pela *Apple* (empresa que a desenvolveu). De acordo com o CEO da *Apple*, Tim Cook, mais de 800 milhões de aparelhos *iOS* – como o *iPhone*, *iPad* e *iPod touch* – já foram vendidos desde o seu lançamento. A plataforma *iOS* é atualizada periodicamente (uma vez ao ano) e a Tabela 5 mostra as versões de *iOS* já lançadas.

¹ <https://play.google.com/store>

Número da versão	Ano Lançamento
iOS 1.x	2007
iOS 2.x	2008
iOS 3.x	2009
iOS 4.x	2010
iOS 5.x	2011
iOS 6.x	2012
iOS 7.x	2013
iOS 8.x	2014

Tabela 5 – Versões da plataforma *Android*

Assim como a plataforma *Android*, a *iOS* também possui um mercado centralizado acessível por qualquer aparelho, chamado de *Apple Store*², para aquisição e publicação de aplicativos.

2.1.3.3 *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 dispositivos móveis, 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*. De acordo com (BUCK, 2012), na plataforma *iOS* a *OpenGL ES* pode ser utilizada por meio do *framework* chamado *GLKit* (introduzido no *iOS* 5). Ele provê classes e funções que simplificam o uso da *OpenGL ES* no contexto do *iOS*.

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.

² <http://store.apple.com/br>

Assim, esta seção apresenta alguns conceitos necessários para o entendimento dos *shaders* implementados.

2.2.1 Renderização *Flat, Gouraud e Phong*

Na área de computação gráfica, *Flat Shading*, *Gouraud Shading* e *Phong Shading* são os *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 que as cores se diferenciem nos vértices de um mesmo polígono, somente uma cor é escolhida entre elas e é aplicada em todo o polígono.

A computação dos cálculos de luz nos vértices seguida por uma interpolação linear dos resultados é denominada 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 da luz em cada vértice e os resultados serão interpolados. Em seguida, o *fragment shader* propaga este valor para as próximas etapas.

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 2.1.

$$I_T = I_A + \sum I_D + I_E \quad (2.1)$$

onde I_T é a iluminação total, I_A é a iluminação ambiente, I_D é a iluminação difusa e I_E é a iluminação especular.

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 2.2, onde I_A é a intensidade de luz ambiente, K_A é o coeficiente de refletividade ambiente da superfície e L_A é a intensidade da componente ambiente da luz.

$$I_A = K_A L_A \quad (2.2)$$

A intensidadede da 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 2.3, onde I_D é a intensidade da reflexão difusa,

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.

$$I_D = K_D L_D (\vec{l} \cdot \vec{n}) \quad (2.3)$$

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 2.4, onde I_E é a intensidade da reflexão especular, 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.

$$I_E = K_S L_S (\vec{r} \cdot \vec{v})^s \quad (2.4)$$

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, os quais não serão renderizados corretamente. A Figura 3 mostra a diferença entre as três técnicas de *shading* aplicadas em uma esfera com uma luz direcional.

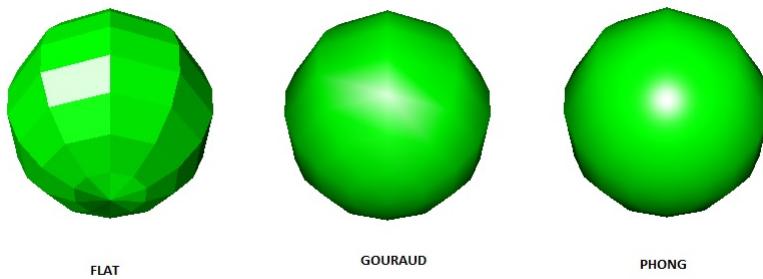


Figura 3 – Comparaçāo entre as técnicas de *shading*

2.2.2 Renderização de Efeito *Cartoon*

A renderização de efeito *cartoon* tem como objetivo emular o efeito de um desenho, que 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 2.5, onde I_L é a intensidade da luz, \vec{l}_d é o vetor da direção da luz e \vec{n} é o vetor normal.

$$I_L = \frac{\vec{l}_d \cdot \vec{n}}{\|\vec{l}_d\| \|\vec{n}\|} \quad (2.5)$$

Como os vetores \vec{l}_d e \vec{n} são normalizados, o cálculo pode ser simplificado de acordo com a Equação 2.6.

$$I_L = \vec{l}_d \cdot \vec{n} \quad (2.6)$$

Após o cálculo da intensidade da luz, ela é mapeada para os tons pré-definidos, como mostra a Figura 4.

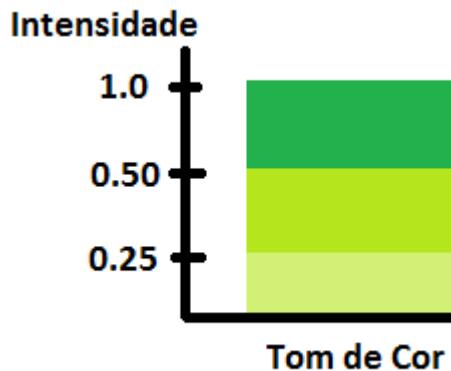
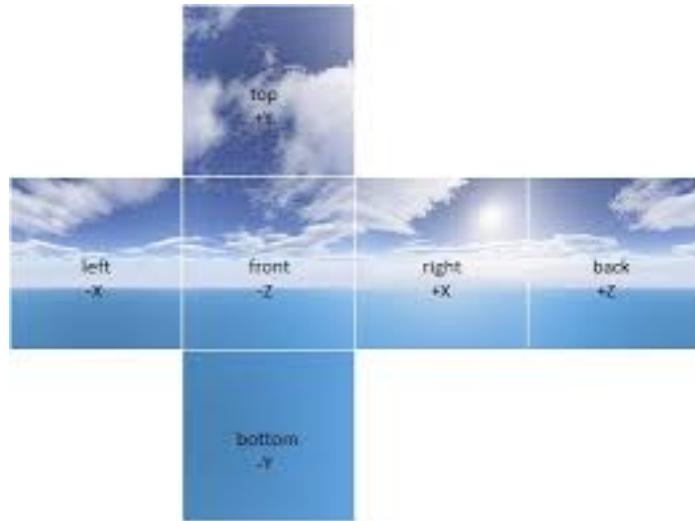


Figura 4 – Exemplo de escolha dos tons

2.2.3 Renderização de Efeito de Reflexão

O efeito de reflexão é obtido através da utilização da técnica chamada *environment mapping*, na qual se reflete uma textura, que é desenhada ao redor do objeto e representa o cenário, chamada de *environment map*. Um exemplo de textura utilizada por esta técnica é apresentada na Figura 5.

Assim, a ideia é obter um vetor que vai da posição da câmera ao objeto e refletí-lo, baseando-se na normal da superfície. Isto gera um outro vetor que é utilizado para determinar a cor, baseando-se na textura (*environment map*).

Figura 5 – *Environment Map*

2.3 Complexidade Assintótica

Complexidade assintótica 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 assintótica 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 assintótica de um código de forma teórica e experimental.

2.3.1 Abordagem Teórica

O cálculo da complexidade assintótica 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 (2.7)

$$y = n^2 + 10n + 1000 \quad (2.7)$$

poderia ser aproximada pela Equação (2.8).

$$y \approx n^2 \quad (2.8)$$

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 6. A Figura 6 mostra uma comparação entre estas curvas.

Complexidade	Descrição
Constante	Ocorre quando as instruções do programa independem do número de elementos de entrada.
$\log N$	Ocorre geralmente em programas que resolvem grandes problemas dividindo-os em partes menores, reduzindo o seu tamanho por uma razão 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 da entrada.
N^3	Ocorre quando o algoritmo é cúbico, ou seja, quando processa todos as triplas da entrada.
2^N	Ocorre quando o algoritmo segue uma função exponencial, ou seja, quando o N dobra o tempo de execução é elevado ao quadrado.

Tabela 6 – Valores mais comuns de complexidade assintótica

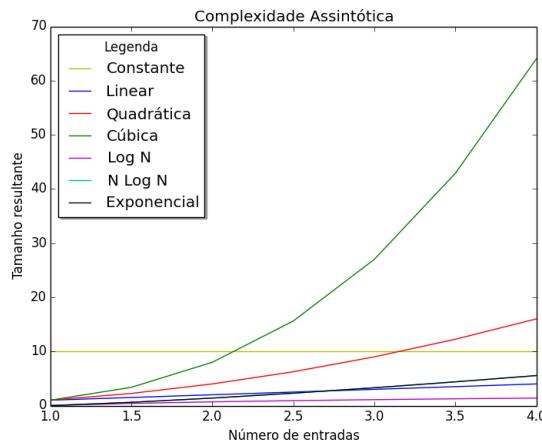


Figura 6 – Comparação da Complexidade Assintótica

2.3.1.1 Notação *Big-O*

A notação *Big-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 $O(n^2)$.

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 Abordagem Experimental

Como não é possível determinar a complexidade assintótica dos *shaders* de forma teórica, pois não estão disponíveis as implementações das funções da API da 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 qual curva melhor aproxima estes pontos. O método dos mínimos quadrados é usado para ajustar um conjunto de pontos (x, y) a uma determinada curva.

2.3.2.1 Ajuste Linear

No caso do ajuste à 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 erros mínimos quadrados aos dados, e a sua solução é dada pela Equação (2.9), em que é possível determinar os coeficientes a e b e, consequentemente, a equação da reta que constitui a aproximação.

$$v = (M^T M)^{-1} M^T y \quad (2.9)$$

onde

$$M = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}, \quad v = \begin{bmatrix} a \\ b \end{bmatrix} \text{ e } y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \quad (2.10)$$

2.3.2.2 Ajuste Exponencial

De acordo com (LEITHOLD, 1994), a função da exponencial pode ser dada como na Equação (2.11), em que e , c , k são constantes (e é a constante neperiana).

$$y = ce^{-kt} \quad (2.11)$$

Aplicando a função logarítmico dos dois lados da equação, obtém-se a Equação (2.12)

$$\ln y = \ln c + \ln e^{-kt} \quad (2.12)$$

que pode ser simplificada na Equação (2.13) (onde \bar{b} é uma nova constante) que equivale à equação da reta.

$$\bar{y} = \bar{a} + \bar{b}t \quad (2.13)$$

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

$$M = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}, \quad y = \begin{bmatrix} \ln y_1 \\ \ln y_2 \\ \vdots \\ \ln y_n \end{bmatrix} \quad (2.14)$$

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

$$c = e^{\bar{a}} \text{ e } \bar{b} = -k \quad (2.15)$$

2.3.2.3 Ajuste para Polinômio de Segundo e Terceiro Grau

O ajuste de segundo grau é parecido com o linear, em que a Equação 2.9 também é utilizada. Porém a matriz M é redefinida para:

$$M = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 \end{bmatrix} \quad (2.16)$$

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

$$M = \begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & x_n^3 \end{bmatrix} \quad (2.17)$$

2.3.2.4 Cálculo dos Erros dos Ajustes

O cálculo dos erros dos ajustes para as funções é calculado a fim de saber qual delas melhor se aproxima ao conjunto de pontos. Estes erros são calculados de acordo com a Equação 2.18, em que cada erro (e_n) está associado à distância do ponto gerado pela curva ajustada e o ponto original.

$$E = \sqrt[2]{e_1 + e_2 + \dots + e_n} \quad (2.18)$$

3 Desenvolvimento

Este capítulo descreve os passos para a realização deste trabalho, mostrando desde os equipamentos utilizados até 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 publicado e desenvolvido na área. Feito isto, 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

O computador utilizado para o desenvolvimento na plataforma *Android* 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. Já o utilizado para desenvolvimento na plataforma *iOS* foi um *Macbook Pro 11.1*, que possui processador *Intel Core i5* de 2,6 GHz e 8 GB de memória RAM.

A Tabela 7 mostra os dispositivos móveis utilizados, que são equipamentos com diferentes resoluções e desempenhos. O aplicativo de *benchmark 3D Mark*¹ foi utilizado para comparar os diferentes desempenhos. Ele roda uma série de testes gráficos com cenas tridimensionais, a fim de estressar o desempenho da *GPU* atribuindo uma pontuação final, em que quanto maior a pontuação, melhor o desempenho.

Dispositivo	Plataforma	Resolução	GPU	Pontuação
<i>Nexus 4</i>	<i>Android</i>	768 x 1280	<i>Adreno 320</i>	7.106
<i>HTC One</i>	<i>Android</i>	1080 x 1920	<i>Adreno 320</i>	10.184
<i>iPad Air</i>	<i>iOS</i>	2048 x 1536	<i>PowerVR G6430</i>	14.952
<i>iPhone 5s</i>	<i>iOS</i>	1136 x 640	<i>PowerVR G6430</i>	14.750

Tabela 7 – Dispositivos móveis utilizados

¹ <http://www.3dmark.com/>

3.3 Configuração do Ambiente e Ferramentas

Em seguida serão apresentadas as configurações dos ambientes de trabalho e as ferramentas utilizadas.

3.3.1 Plataforma *Android*

Uma das alternativas para o desenvolvimento em plataforma *Android* foi utilizar a ferramenta *Eclipse*² (Figura 7), que é um ambiente de desenvolvimento integrado (*Integrated Development Environment – IDE*) *open source*. Adicionalmente, foi preciso instalar o *Android Software Development Kit*³ e o *plugin ADT* (*Android Development Tools*)⁴, que permitem desenvolver e depurar aplicações pra *Android*. Outra alternativa possível seria utilizar o *Android Studio*⁵, lançado recentemente (2013) pela empresa *Google*, que já possui todos os pacotes e configurações necessárias para o desenvolvimento, incluindo o *Software Development Kit* (SDK), as ferramentas e os emuladores.

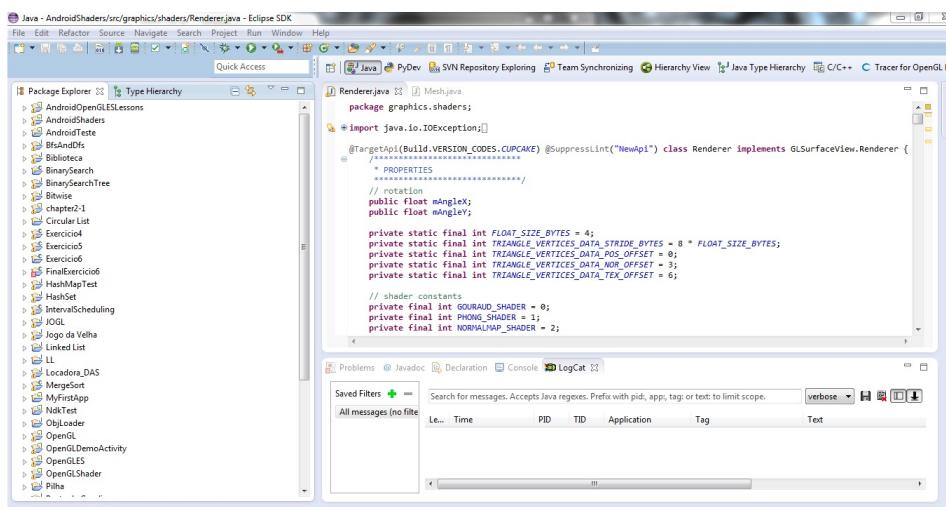


Figura 7 – Ambiente de desenvolvimento *Eclipse*

Além disso, também foi necessário instalar o *Android NDK*, a fim de poder utilizar linguagem de código nativo C e acessar as extensões da *OpenGL ES*. A biblioteca gráfica para sistemas embarcados *OpenGL ES* já faz parte das ferramentas de desenvolvimento da plataforma *Android*.

3.3.2 Plataforma *iOS*

Para o desenvolvimento na plataforma *iOS*, foi utilizada a ferramenta *Xcode*. Segundo (NEUBURG, 2013), um novo projeto no *Xcode* já vem com todos os arquivos e

² <http://www.eclipse.org/>

³ <http://developer.android.com/sdk/index.html>

⁴ <http://developer.android.com/tools/sdk/eclipse-adt.html>

⁵ <http://developer.android.com/sdk/installing/studio.html>

configurações necessários para o desenvolvimento de um aplicativo (Figura 8). Esta ferramenta também possui um módulo chamado *Instruments*, que analisa o comportamento interno de um aplicativo graficamente e numericamente, sendo possível monitorar diversas métricas.

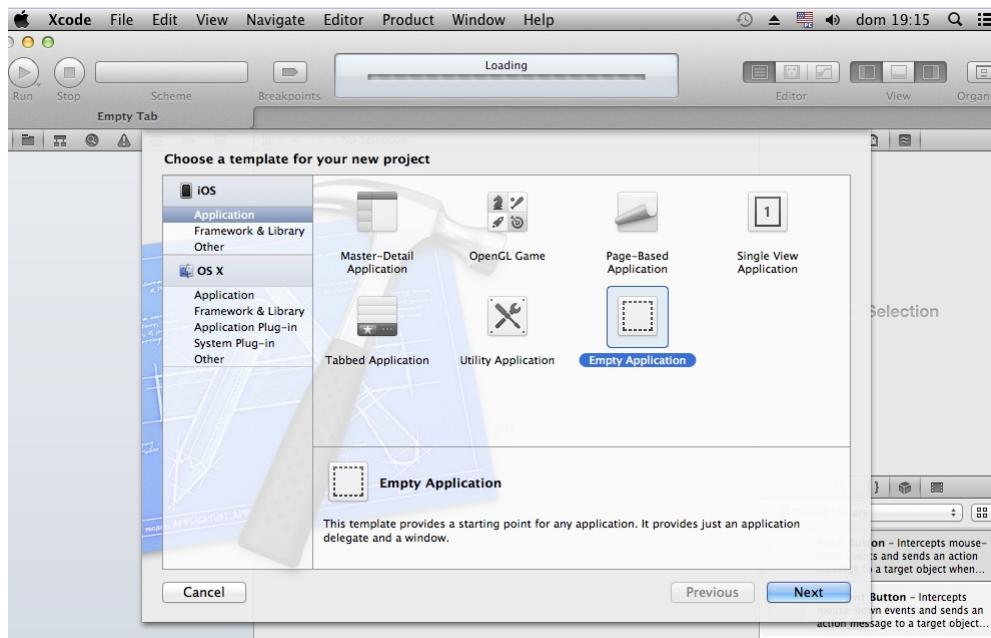


Figura 8 – Novo projeto de um aplicativo na ferramenta *Xcode*

3.3.3 Modelagem dos Objetos Tridimensionais

A fim de variar a contagem poligonal dos objetos tridimensionais, foi utilizada a ferramenta de modelagem tridimensional *open source* chamada Blender⁶, a qual está ilustrada na Figura 9. 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).

3.3.4 Coleta de Medições

Para a coleta das medições quanto a todo o processo de renderização, na plataforma *Android*, foi utilizada uma extensão da *OpenGL ES*. Esta extensão se chama `GL_EXT_disjoint_timer_query`⁷, e dentre os dispositivos utilizados neste trabalho, só está disponível para o *Nexus 4* a partir da versão de *Android 4.4 (KitKat)*. A fim de 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

⁶ <http://www.blender.org/>

⁷ <http://www.khronos.org/registry/gles/>

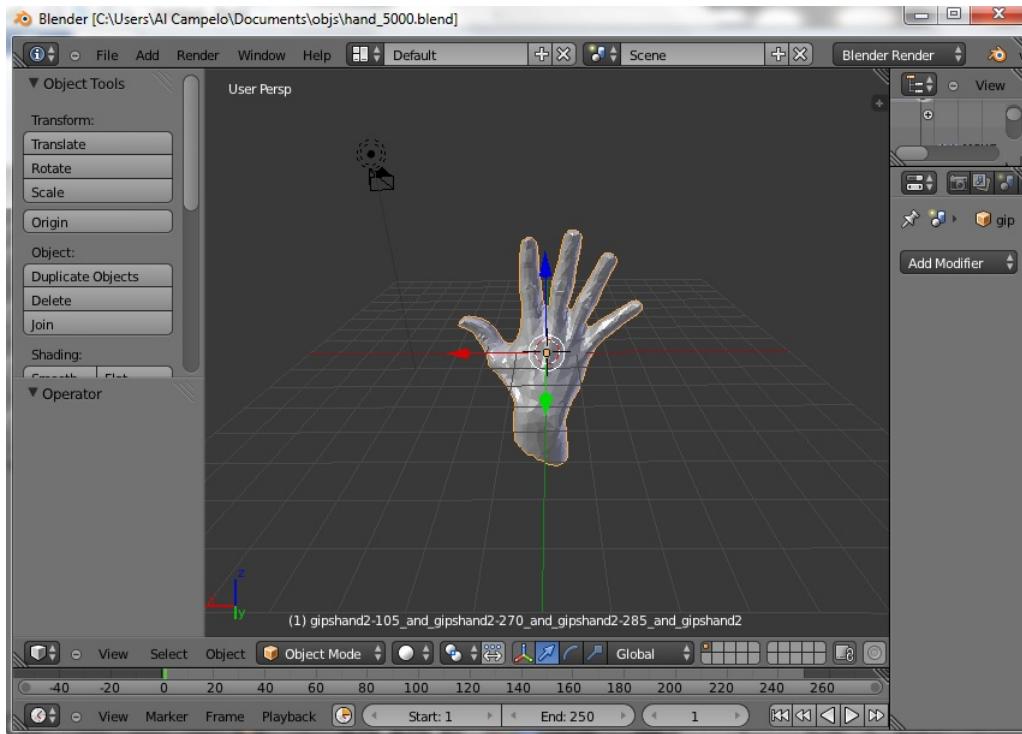


Figura 9 – Ferramenta de modelagem tridimensional

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*).

A fim de realizar a mesma medição na plataforma *iOS*, utilizou-se (como mostra a Figura 10) o módulo *Instruments* da ferramenta *Xcode*.

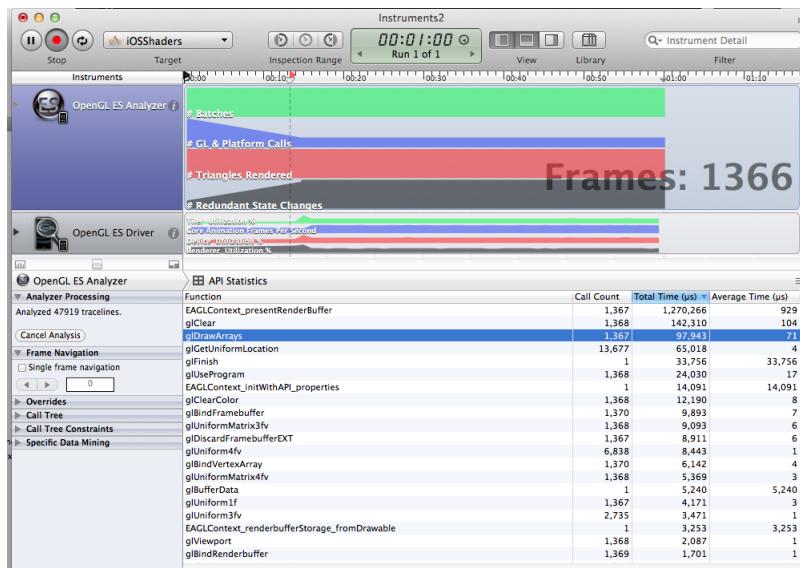


Figura 10 – Analisador de *OpenGL ES*

Para a coleta das medições específicas do *vertex* e *fragment shaders* foi utilizada a ferramenta *Adreno Profiler*, pois os celulares com plataforma *Android* utilizados possuem a GPU *Adreno*. Quanto à plataforma *iOS*, não é possível coletar estas medições.

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 11, 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 de também gerar outras estatísticas.

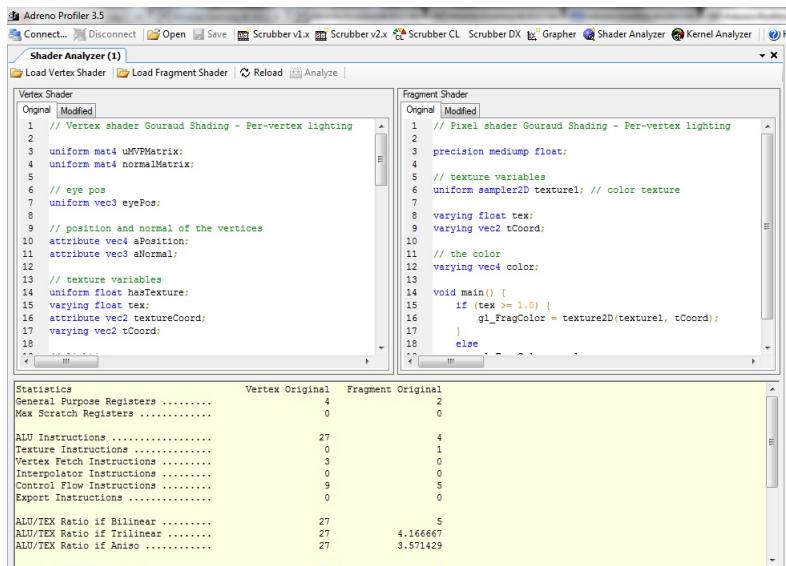


Figura 11 – Ferramenta *Adreno Profiler*: analisador de *shaders*

O módulo gráfico desta ferramenta permite analisar algumas métricas, relacionadas ao *vertex* e *fragment shaders*, conforme ilustrado na Figura 12, em que 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.



Figura 12 – Ferramenta *Adreno Profiler*: visualização de métrica quadros por segundo

3.3.5 Automatização de Cálculo e Plotagem

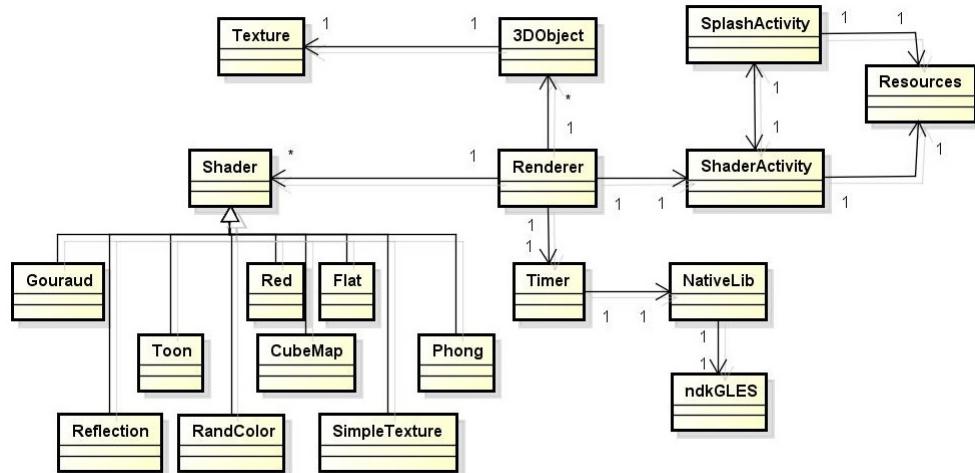
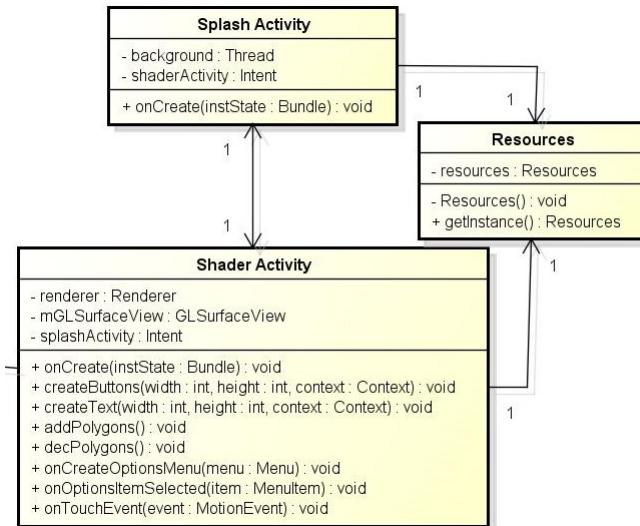
Para a automatização da plotagem dos gráficos e aplicação do método dos mínimos quadrados (descrito na Seção 2.3.2) utilizou-se a linguagem de programação Python⁸ versão 2.7.3, juntamente com os pacotes *matplotlib*⁹ e *numpy*¹⁰.

3.3.6 Controle de Versionamento

O controle de versionamento do código foi feito por meio do sistema de controle de versão Git¹¹ utilizando o *forge GitHub*¹². Foram criados repositórios tanto para as implementações em *Android* e *iOS*, quanto para a automatização da análise da complexidade assintótica.

3.4 Implementação na Plataforma *Android*

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 13) 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.

Figura 13 – Diagrama de Classe da Implementação em *Android*Figura 14 – Detalhamento das classes *Shader Activity*, *Splash Activity* e *Resources*

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 14), a *Shader* e a *Splash*.

A *Splash Activity* (Figura 15) é responsável pela visualização da tela de *loading*

⁸ <http://www.python.org.br/>

⁹ <http://www.matplotlib.org/>

¹⁰ <http://www.numpy.org/>

¹¹ <http://git-scm.com/>

¹² <https://github.com/>

enquanto são carregados 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 uma *thread*. Estes recursos são gerenciados pela classe *Resources*, que utiliza o padrão de projeto *Singleton*, que garante a existência de apenas uma instância da classe, que será acessada posteriormente.

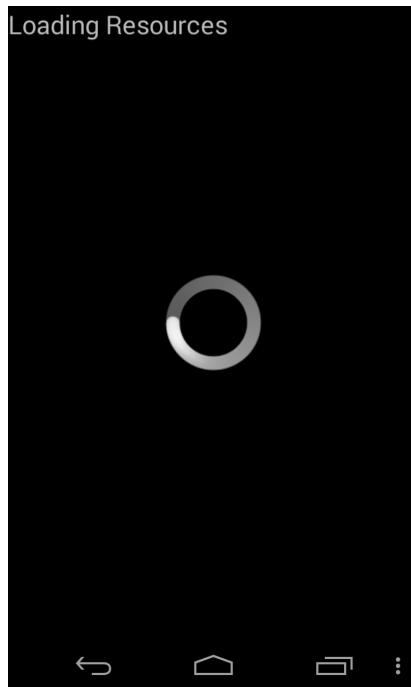
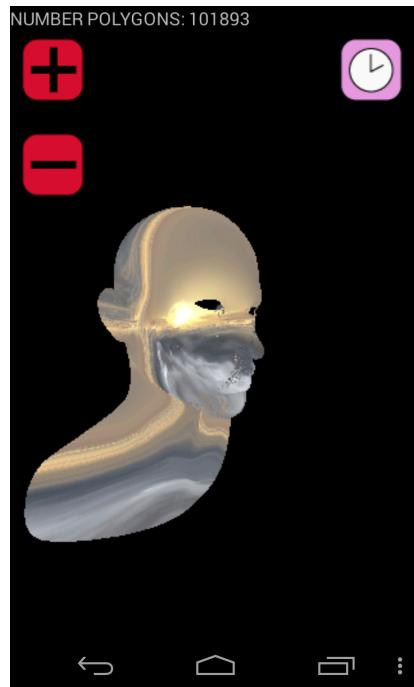
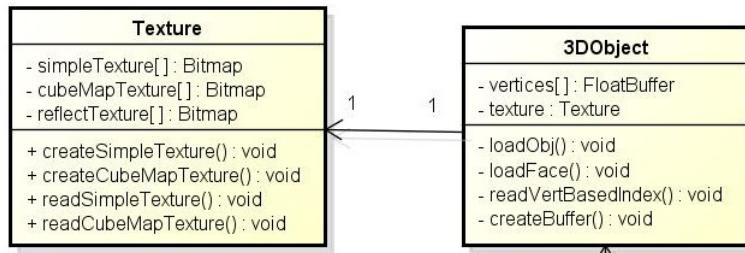


Figura 15 – Tela da *Splash Activity*

A *Shader Activity* (Figura 16) 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 o menu que troca de *shader*, os botões que aumentam ou diminuem o número de polígonos, a informação do tempo de renderização e a da quantidade de polígonos. O aumento do número de polígonos é realizado através da troca de objetos que possuem arquivos *obj* diferentes, que já foram carregados pela *Splash 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 renderizados.

Figura 16 – Tela da *Shader Activity*Figura 17 – Detalhamento das classes *3DObject* e *Texture*

3.4.2 Objeto Tridimensional

O objeto tridimensional foi representado pela composição das classes *3DObject* e *Texture*. A classe *3DObject*, mostrada na Figura 17, é responsável por ler e interpretar os arquivos *obj* (descritos no Anexo B), criados por meio da ferramenta *Blender*.

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 são renderizados. Neste *buffer* cada coordenada relacionada a um vértice (posição, normal e textura) é armazenada alternadamente, como mostra a Figura 18.

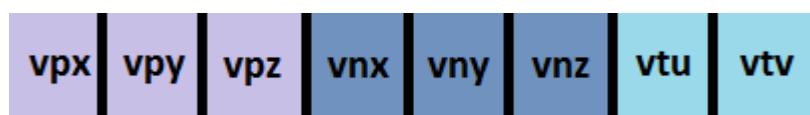


Figura 18 – Ordem das coordenadas de posição, normal e textura para um vértice

A classe *Texture* gera as texturas utilizadas pelos *shaders SimpleTexture*, *CubeMap* e *Reflection*, a partir de imagens. Estas 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 19). Como a orientação do eixo de coordenadas *y* da ferramenta *Blender* é diferente da *OpenGL 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.

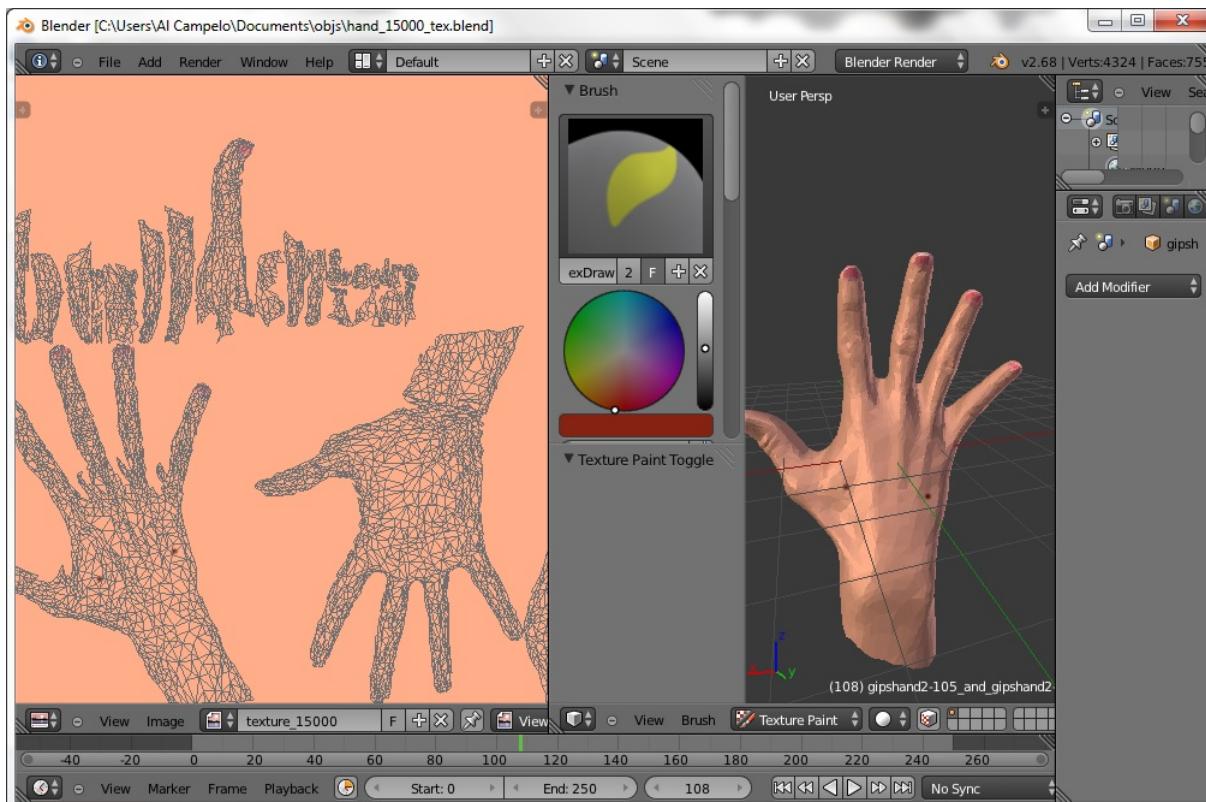


Figura 19 – Técnica de mapeamento de textura utilizada para cada modelo 3D

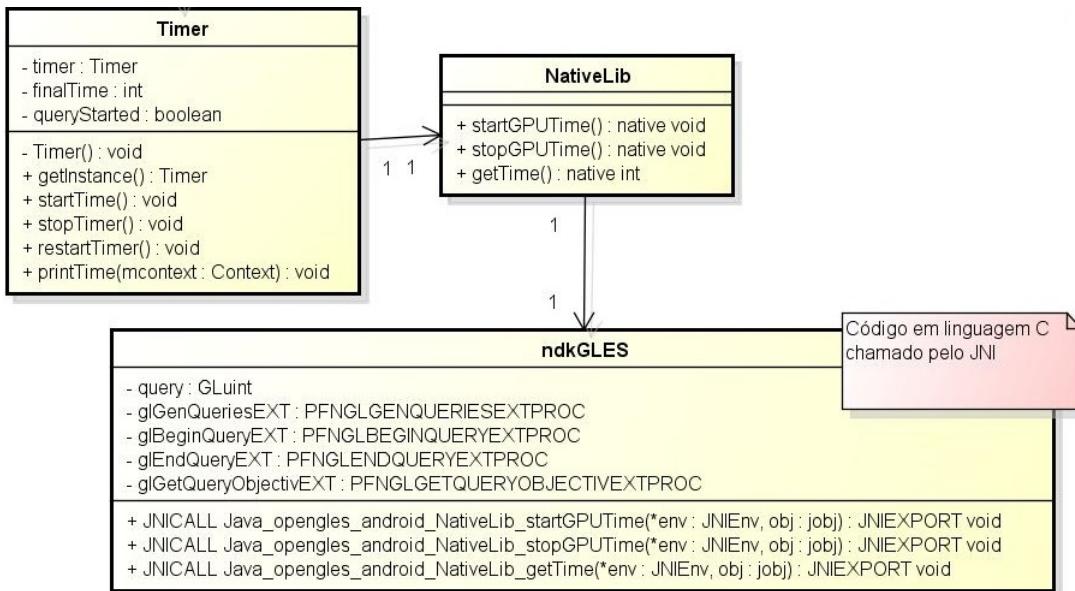
A Figura 20 apresenta a imagem resultante da técnica de mapeamento realizada.

3.4.3 Cálculo do Tempo de Renderização

A Figura 21 detalha a classe *Timer*, que realiza a média de dez medições do tempo de renderização (em nanosegundos) para cada objeto tridimensional. Cada medição é feita utilizando a linguagem C e a extensão da *OpenGL ES*, citada na Seção 3.6.1, chamada `GL_EXT_disjoint_timer_query`. A integração entre o código em linguagem C e o código



Figura 20 – Textura gerada a partir da técnica de mapeamento

Figura 21 – Detalhamento das classes *Timer* e *NativeLib*

em Java é feita por meio da classe *NativeLib*. Caso a extensão não esteja disponível para o dispositivo, um alerta é emitido.

3.4.4 Renderização

A Figura 22 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 implementa as funções da biblioteca *OpenGL ES* *onSurfaceCreated()*, *onDrawFrame()* e *onSurfaceChanged()*. A primeira função é chamada apenas uma vez quando a *view* da

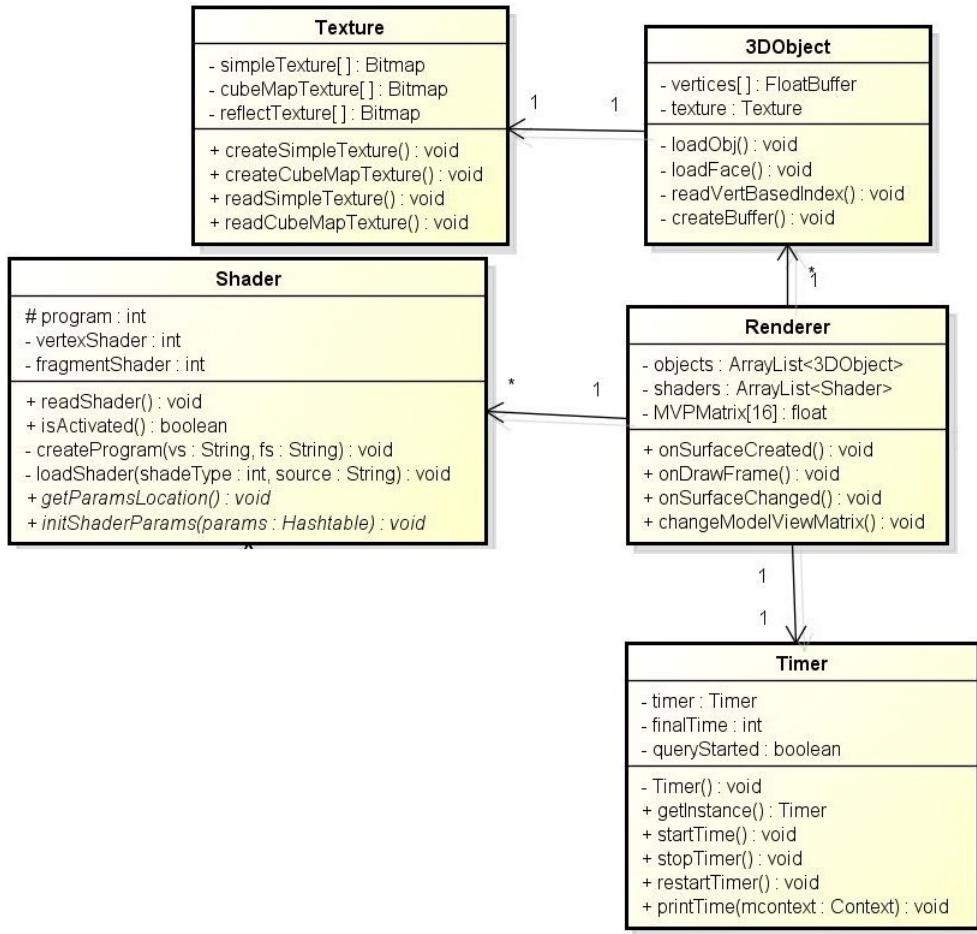


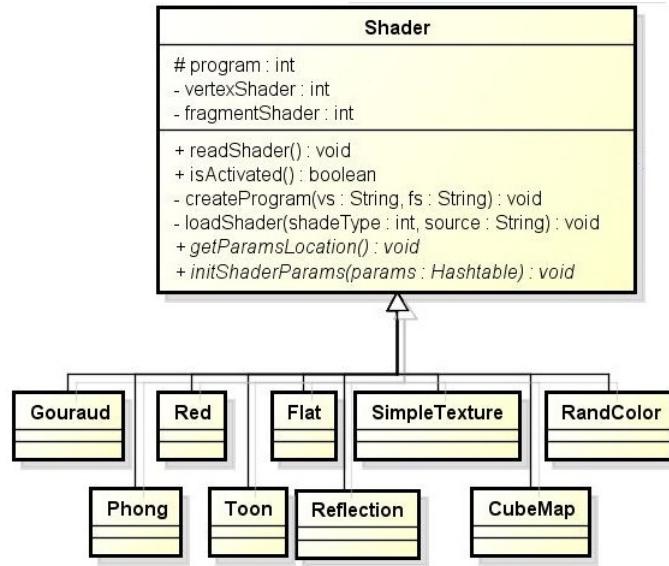
Figura 22 – Detalhamento da classe *Renderer*

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*, onde é feita a renderização por meio da função `glDrawArrays()`.

3.4.5 Shaders

A classe *Shader* (Figura 23) 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)`. 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 24.

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

Figura 23 – Detalhamento da classe *Shader*

3.4.5.1 *Phong Shader*

O *vertex* e *fragment shaders* do *phong shading* implementam a técnica descrita na Seção 2.2.1, 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 vetores 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 Gouraud Shader

O *Gouraud Shader*, assim com o *Phong Shader*, também implementa a técnica descrita na Seção 2.2.1, porém os cálculos de luz e da cor são feitos no *vertex shader* para serem interpolados e passados para o *fragment shader*, como mostram o Código 3.3 e o Código 3.4.

Código 3.3 – Gouraud Shader: vertex shader

```
1 uniform mat4 uMVPMatrix;
2 uniform mat4 normalMatrix;
3 // eye pos
4 uniform vec3 eyePos;
5 // position and normal of the vertices
6 attribute vec4 aPosition;
7 attribute vec3 aNormal;
8 // lighting
9 uniform vec4 lightPos;
10 uniform vec4 lightColor;
11 // material
12 uniform vec4 matAmbient;
13 uniform vec4 matDiffuse;
14 uniform vec4 matSpecular;
15 uniform float matShininess;
16 // color to pass on
17 varying vec4 color;
18
19 void main() {
20     // eyePos
21     vec3 eP = eyePos;
22     vec4 nm = normalMatrix * vec4(aNormal, 1.0);
23     // normal
24     vec3 EyespaceNormal = vec3(uMVPMatrix * vec4(aNormal, 1.0));
25     // the vertex position
26     vec4 posit = uMVPMatrix * aPosition;
27     // light dir
28     vec3 lightDir = lightPos.xyz - posit.xyz;
29     vec3 eyeVec = -posit.xyz;
30     vec3 N = normalize(EyespaceNormal);
31     vec3 E = normalize(eyeVec);
32     vec3 L = normalize(lightDir);
33     // Reflect the vector
34     vec3 reflectV = reflect(-L, N);
35     // Get lighting terms
36     vec4 ambientTerm;
37     ambientTerm = matAmbient * lightColor;
38     vec4 diffuseTerm = matDiffuse * max(dot(N, L), 0.0);
39     vec4 specularTerm = matSpecular *
40         pow(max(dot(reflectV, E), 0.0), matShininess);
41     color = ambientTerm + diffuseTerm + specularTerm;
```

```

42
43     gl_Position = uMVPMatrix * aPosition;
44 }

```

Código 3.4 – *Gouraud Shader: fragment shader*

```

1 precision mediump float;
2 // the color
3 varying vec4 color;
4
5 void main() {
6     gl_FragColor = color;
7 }

```

3.4.5.3 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 3.5.

Código 3.5 – *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 3.6) estabelece que todo fragmento possui a cor vermelha, por meio da variável pré-definida *gl_FragColor*.

Código 3.6 – *Red Shader: fragment shader*

```

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

```

3.4.5.4 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 3.7 mostra o cálculo da intensidade da luz por vértice, utilizando 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 3.7 – 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 3.8).

Código 3.8 – 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 }

```

3.4.5.5 *Flat Shader*

A ideia do *Flat Shader* é tornar um modelo tridimensional em bidimensional, achato. Para isto, a coordenada *z* deve ser definida como zero, como é mostrado no Código 3.9. O *fragment shader* (Código 3.10) apenas define uma cor para o fragmento (Código 3.10).

Código 3.9 – Flat Shader: *vertex shader*

```

1 uniform mat4 uMVPMatrix;
2 attribute vec4 aPosition;
3

```

```

4 void main()
5 {
6
7     vec4 v = aPosition;
8     v.z = 0.0;
9     gl_Position = uMVPMatrix * v;
10
11 }
```

Código 3.10 – *Flat Shader: fragment shader*

```

1
2 void main()
3 {
4     gl_FragColor = vec4(0.82, 0.50, 0.20, 1.0);
5 }
```

3.4.5.6 Random Color Shader

O *Random Color Shader* determina a cor do fragmento, baseando-se em um cálculo matemático, que determina a cor final aleatoriamente. Este cálculo da cor é feito no *vertex shader* (Código 3.11) e o valor resultante é passado para o *fragment shader*, por meio da variável *color* do tipo *varying*. Cada componente da cor é calculada passando uma coordenada (*x*, *y* ou *z*) para a função `random(vec2 v)`, que retorna um valor aleatório para a componente da cor, baseando-se nesta coordenada. O Código 3.12 apenas determina a cor do fragmento, calculada pelo *vertex shader*.

Código 3.11 – *Random Color Shader: vertex shader*

```

1 uniform mat4 uMVPMatrix;
2 attribute vec4 aPosition;
3 varying vec4 color;
4
5 float random( vec2 v){
6     // e^pi (Gelfond's constant)
7     const vec2 r = vec2(23.1406926327792690,
8     // 2^sqrt(2) (Gelfond Schneider constant)
9                         2.6651441426902251);
10 return mod( 123456789., 1e-7 + 256. * dot(v,r) );
11 }
12
13 void main()
14 {
15     vec2 r = vec2(aPosition.x, aPosition.z);
16     vec2 g = vec2(aPosition.y, aPosition.x);
```

```

17     vec2 b = vec2(aPosition.z, aPosition.y);
18     color = vec4(random(r),random(g),random(b),1.0);
19     gl_Position = uMVPMatrix * aPosition;
20
21 }
```

Código 3.12 – *Random Color Shader: fragment shader*

```

1 varying mediump vec4 color;
2
3 void main()
4 {
5     gl_FragColor = color;
6 }
```

3.4.5.7 Simple Texture Shader

O *vertex shader* do *simple texture shading* primeiramente armazena as coordenadas de textura numa variável do tipo *varying* (Código 3.13), e as repassa para o *fragment shader*, além de também definir a posição do vértice.

Código 3.13 – *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 3.14, 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()` da GLSL.

Código 3.14 – *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 }
```

3.4.5.8 CubeMap Shader

O *vertex shader* do *CubeMap Shader* é simples e só define a posição do vértice (Código 3.15).

Código 3.15 – *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 3.16), por sua vez, utiliza a função `textureCube(samplerCube s, vec3 coord)` da GLSL, que recebe como parâmetros o vetor normal e a textura a ser mapeada.

Código 3.16 – *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 }
```

3.4.5.9 Reflection Shader

O *Reflection Shader* implementa a técnica descrita na Seção 2.2.3. O seu *vertex shader* é responsável por indicar que a posição do vértice se dá pela multiplicação da coordenada (variável `vec4 aPosition`) pela matriz de projeção, visualização e modelagem, como é mostrado no Código 3.17. Além disso, no *vertex shader* são declarados dois vetores do tipo *varying* (que passa o valor da variável ao *fragment shader*) que estão relacionados com o vetor de direção da câmera e o vetor normal.

Código 3.17 – *Reflection Shader: vertex shader*

```

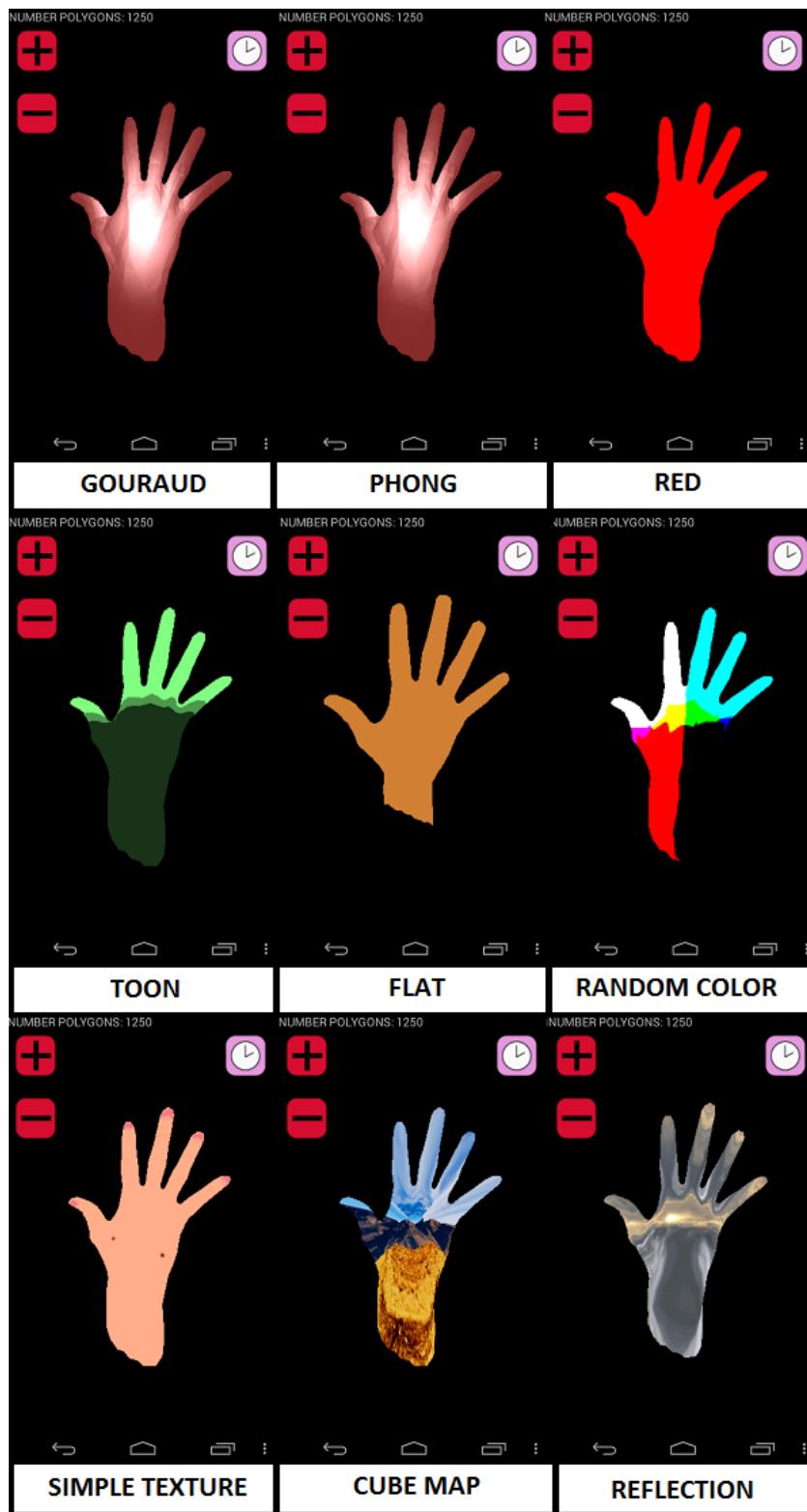
1 attribute vec4 aPosition;
2 attribute vec3 aNormal;
3
4 varying vec3 EyeDir;
```

```
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`. O vetor da direção da reflexão é utilizado na função `textureCube(samplerCube s, vec3 coord)` (Código 3.18), em que determina-se a cor do fragmento, baseando-se nesta direção e em uma imagem.

Código 3.18 – *Reflection Shader: fragment shader*

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

Figura 24 – *Shaders* Implementados

3.5 Implementação na Plataforma iOS

A estrutura do programa portado para a plataforma *iOS* é similar à feita para plataforma *Android*, como mostra a Figura 25. Segundo (NEUBURG, 2013), o desenvolvimento para plataforma *iOS* segue o padrão MVC (*Model-View-Controller*), que é um padrão arquitetural. A classe (*RendererViewController*) é a controladora responsável pela integração entre as classes *Shader*, *3DObject* e a classe de visualização *RendererView*. É na controladora que são feitas as principais chamadas da *OpenGL ES*. A classe *3DObject* faz a interpretação do arquivo *obj* para o formato aceito pela *OpenGL ES* e a classe *Shader*, assim como na plataforma *Android*, é responsável por ler, fazer o *attach* e o *link* do *vertex* e do *fragment shaders*.

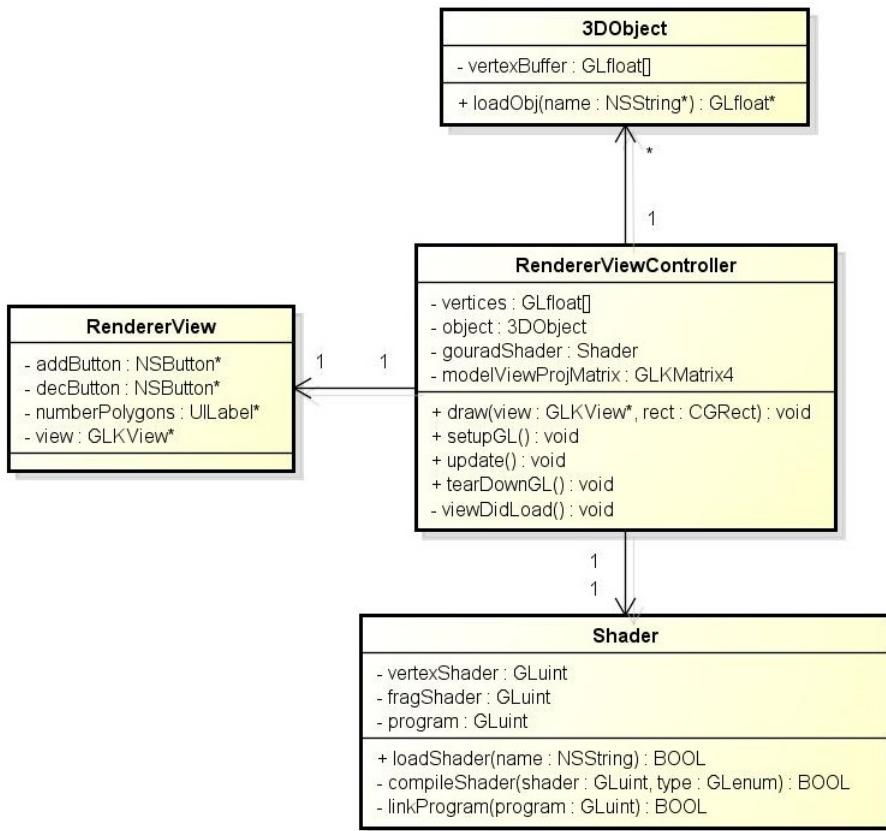


Figura 25 – Diagrama de Classes da Implementação em *iOS*

Assim, escolheu-se um *shader*, no caso o *Gouraud Shader*, para ser implementado e posteriormente realizar as comparações entre os diferentes dispositivos e plataformas. O resultado da implementação pode ser visto na Figura 26.



Figura 26 – *Gouraud Shader* na plataforma *iOS*

3.6 Estimativa Experimental da Complexidade Algorítmica

A estimativa experimental da complexidade assintótica foi realizada 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). Cada um destes processos é explicado a seguir.

3.6.1 Medição do Processo de Renderização

A fim de estimar experimentalmente a análise da complexidade assintótica dos *shaders* implementados, primeiro buscou-se coletar uma métrica relacionada ao tempo de renderização, que calcula o tempo necessário para processar o *shader* como um todo. Por meio de pesquisa e consulta na documentação da *OpenGL ES* para a plataforma *Android*, conseguiu-se encontrar uma extensão desta biblioteca gráfica, explicada na Seção 3.3.4, que permite contabilizar o tempo, em nanosegundos, necessário para realizar chamadas de *OpenGL ES* específicas. Porém esta extensão só está disponível para alguns dispositivos e foi utilizada para medir o tempo de todo o processo de renderização realizado pela função `glDrawArrays()`.

Para a plataforma *iOS*, utilizou-se um módulo da ferramenta *Xcode* também explicado na Seção 3.3.4, que assim como a extensão da *OpenGL ES*, consegue calcular o tempo da chamada da função `glDrawArrays()`, porém em microssegundos.

Assim, as medições do tempo do processo de renderização foram coletadas para o dispositivo *Nexus 4*, *iPhone 5s* e *iPad Air*. A medição não foi feita para o dispositivo *HTC One*, onde a extensão `GL_EXT_disjoint_timer_query` não está disponível.

3.6.2 Medição do *Vertex* e *Fragment Shaders*

Para a coleta de medições relacionadas ao *vertex* e *fragment shaders*, utilizou-se a ferramenta *Adreno Profiler*, mostrada na Seção 3.3.4. 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 exportados no formato CSV. Porém, como esta ferramenta só pode ser utilizada em dispositivos com GPU Adreno, as medições foram realizadas apenas para os dispositivos *Nexus 4* e *HTC One* (para este último dispositivo, escolheu-se apenas o *Gouraud Shader* para comparação).

A medição do *vertex* e *fragment shaders* não puderam ser feitas para a plataforma *iOS*, pois o módulo *Instruments* não disponibiliza nenhuma métrica relacionada.

3.6.3 Plotagem

Após feitas as medições, foram plotados os gráficos tanto para todo o processo de renderização, quanto para o *vertex* e *fragment shaders*. O primeiro conjunto de gráficos está relacionado com o tempo em nanosegundos *versus* a quantidade de polígonos, e o segundo, com o número de instruções por vértice (ou fragmento) *versus* a quantidade de polígonos.

3.6.4 Automatização dos Ajustes das Curvas

A fim de ajustar as curvas obtidas por meio das medições, e consequentemente estimar a complexidade assintótica, utilizou-se o método dos mínimos quadrados (Seção 2.3.2) para funções lineares, quadráticas, cúbicas e exponenciais, calculando-se os respectivos erros, e determinando qual destas curvas melhor aproximava das medições.

Para automatizar este cálculo do ajuste, foi feito um programa em Python que lê os arquivos CSV, computa 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. Este programa 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. Dois exemplos de linhas de comando podem ser vistos no Código 3.19). A primeira linha

está relacionada com o cálculo para todo o processos de renderização, e a segunda, apenas para o *vertex* e *fragment shaders*.

Código 3.19 – Linhas de comando

```
$ python shaderComplexity.py gouraud render_time
$ python shaderComplexity.py gouraud vertex_fragment
```

O programa foi estruturado de acordo com a Figura 27, em que a classe *ReadCSV* é responsável por ler os arquivos CSV e computar 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 ou 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.

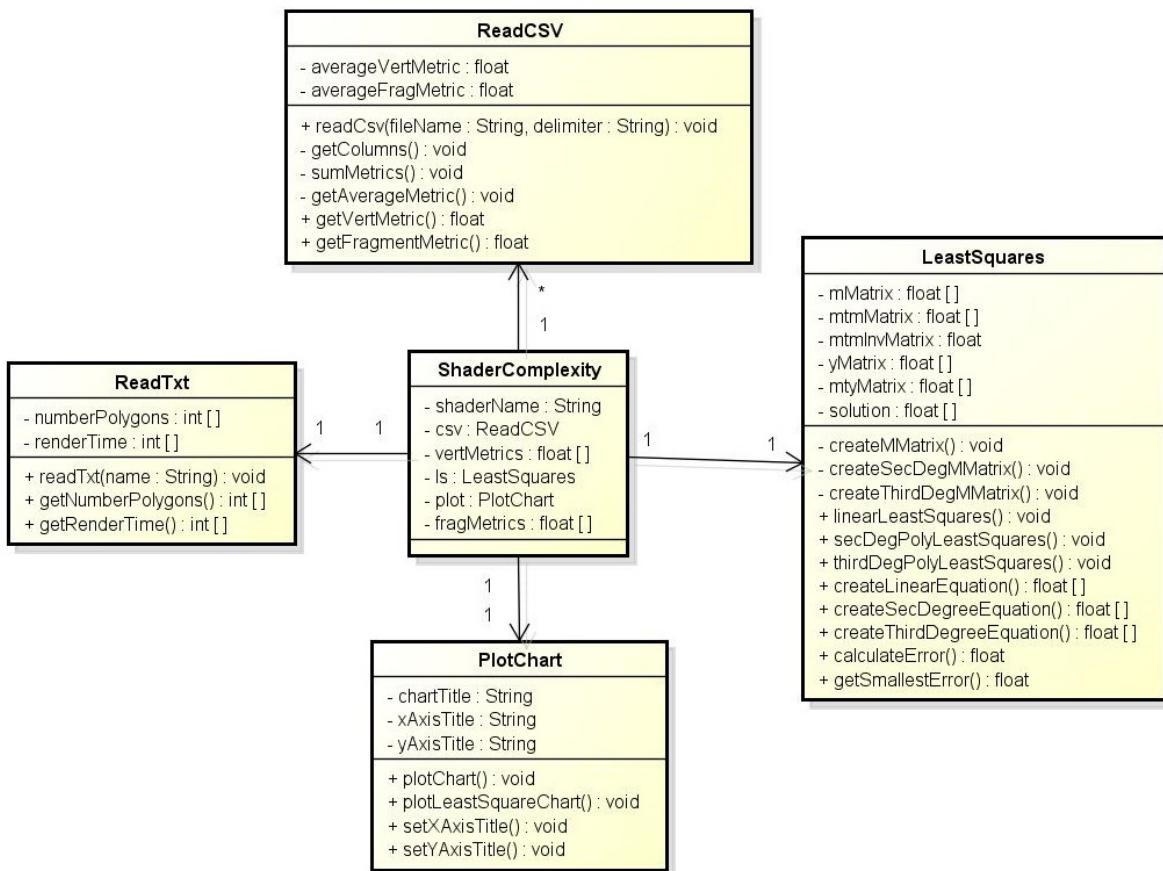


Figura 27 – Diagrama de Classes do *script* de automatização

4 Resultados

Para cada *shader* foram plotados os gráficos das métricas relacionadas ao *vertex* e ao *fragment shaders* e de todo o processo de renderização para diferentes dispositivos. Após as plotagens, percebeu-se que para todos os *shaders* de todos os dispositivos, todos os gráficos relacionados ao *vertex shader* possuíam curvas semelhantes. O mesmo ocorreu para o *fragment shader* e para todo o processo de renderização.

4.1 Dispositivos *Android*

Com o dispositivo *Nexus 4* foi possível plotar os gráficos tanto para todo o processo de renderização, quanto para o *vertex* e *fragment shaders*. Assim, foi visto que os gráficos de todos os *shaders* relacionados ao vértice resultaram em uma função linear (diferindo na inclinação) e os relacionados ao fragmento e todo o processo de renderização deram curvas de formato semelhantes, embora não fosse possível determinar a curva exatamente apenas pela inspeção visual. A Figura 28 e a Figura 29 mostram os gráficos plotados, com relação ao *vertex* e *fragment shaders* de cada *shader* implementado, que demonstram a semelhança destas curvas. A Figura 30 também mostra esta semelhança para todo o processo de renderização de todos os *shaders*.

A Figura 31 e a Figura 32 faz uma comparação entre *shaders*, em que todos os *vertex* e *fragment shaders* são plotados em um mesmo gráfico, assim como as curvas relacionadas a todo o processo de renderização.

Assim, os ajustes destas curvas desconhecidas em relação às curvas pré-definidas (linear, exponencial, polinômios de segundo e terceiro graus) também foram calculados e plotados para cada *shader* (Figura 33 e Figura 34, referentes ao *Reflection Shader*). Também foram determinados os menores erros associados, a fim de descobrir qual curva melhor se ajustava às medições do *fragment shader* e de todo o processo de renderização. Pela a análise do menor erro, calculado de acordo com a Seção 2.3.2, todos os *shaders* se aproximaram melhor de uma curva de terceiro grau, tanto para o *fragment shader*, quanto para todo o processo de renderização.

Com o dispositivo *HTC One* foi possível apenas realizar o cálculo para o *vertex* e *fragment shaders*, pois ele não dá suporte à extensão da *OpenGL ES* utilizada para a coleta do tempo do processo de renderização. A Figura 35 mostra os gráficos plotados para o *vertex* e *fragment shaders* do *Gouraud Shader* (*shader* exemplo utilizado para comparação), em que os formatos são semelhantes aos obtidos com o *Nexus 4*. Assim como no *Nexus 4*, o *vertex shader* também teve comportamento linear e para o *fragment*

shader, a melhor aproximação foi uma curva de terceiro grau (Figura 36).

A Figura 37 mostra as curvas dos *shaders* com relação ao *Nexus 4* e ao *HTC One* plotadas em um mesmo gráfico.

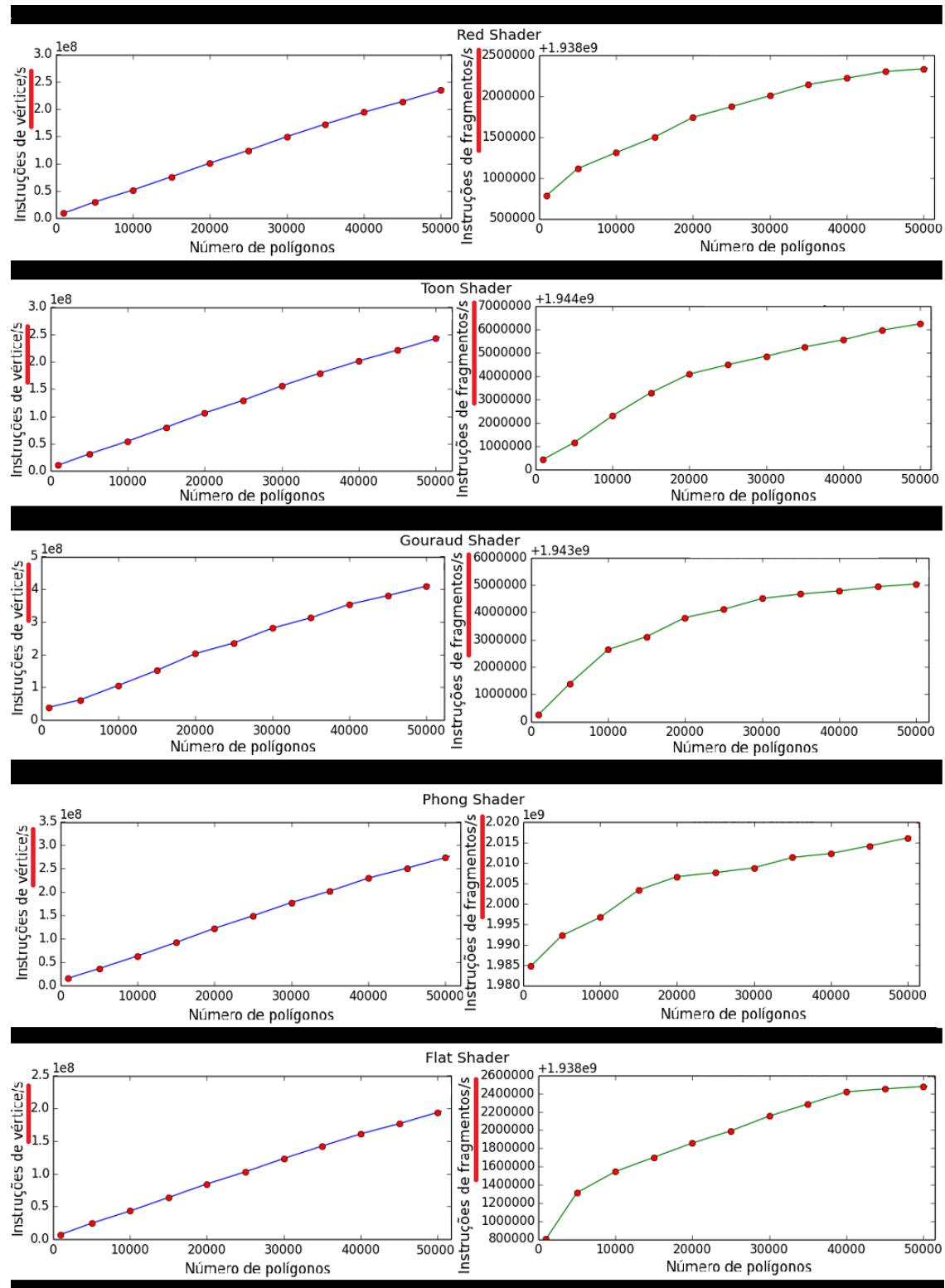


Figura 28 – Gráficos: *Red Shader*, *Toon shader*, *Gouraud Shader*, *Phong Shader* e *Flat Shader* para o *Nexus 4*

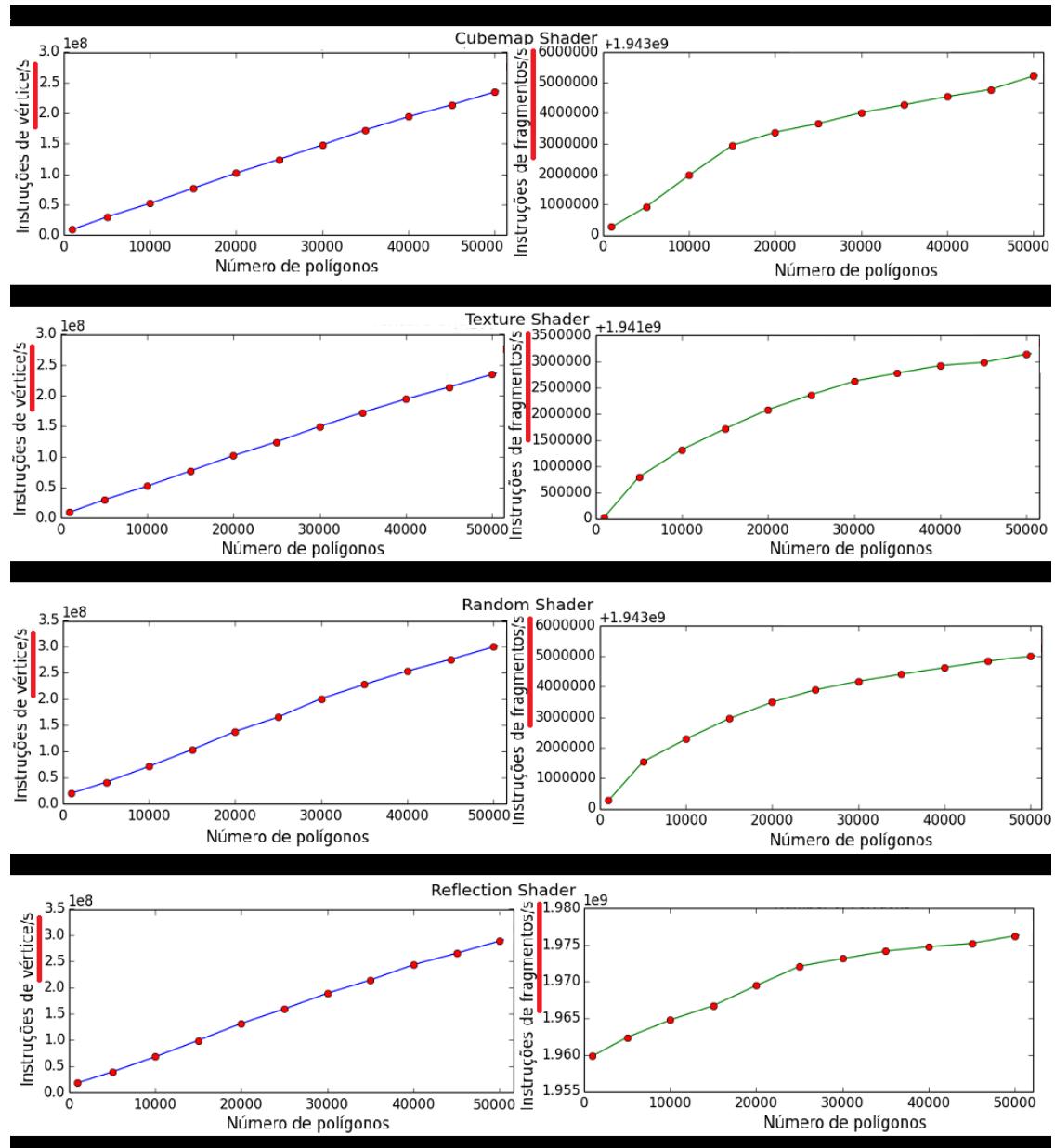


Figura 29 – Gráficos: *Cubemap Shader*, *Texture Shader*, *Random Shader*, *Reflection Shader* para o *Nexus 4*

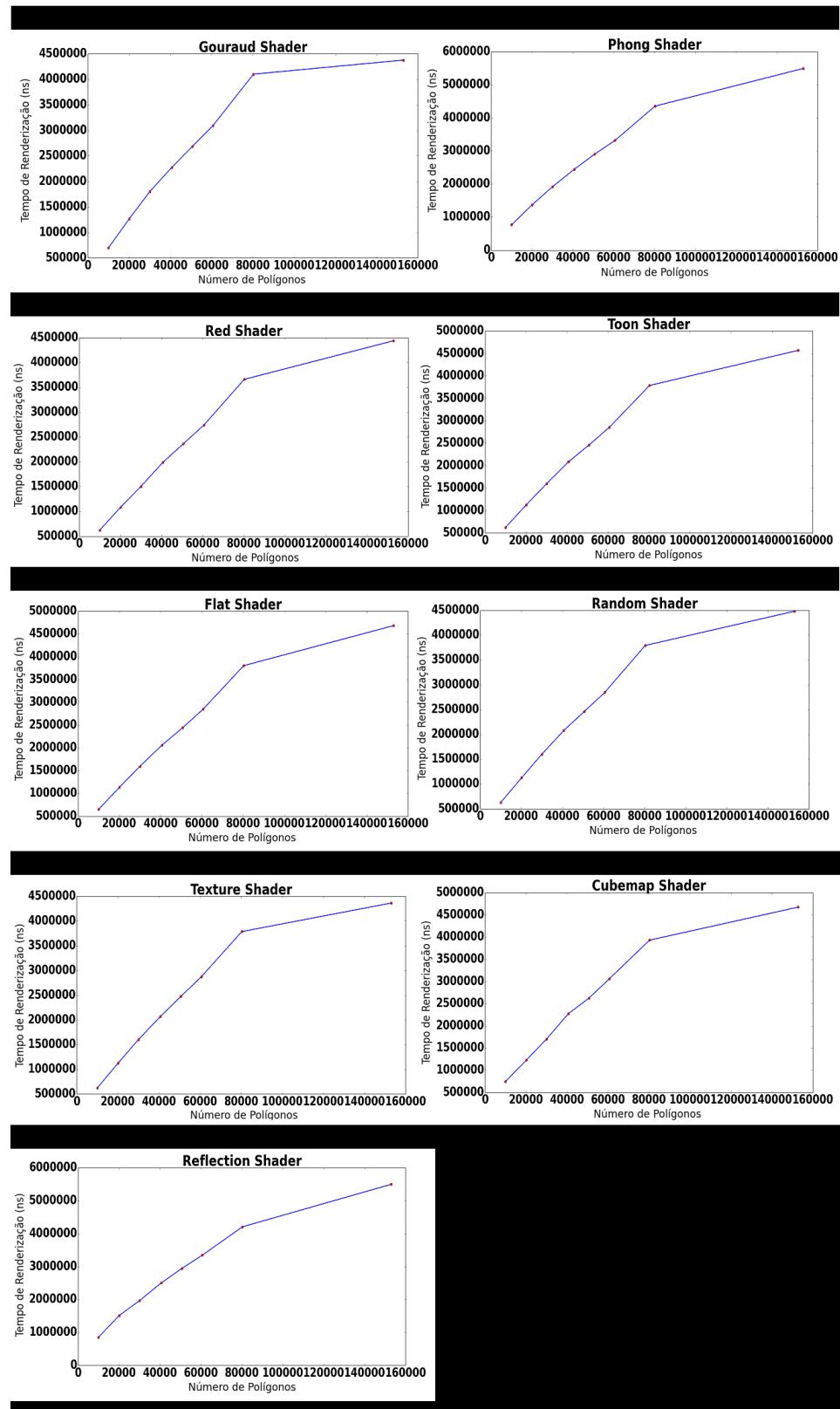


Figura 30 – Gráficos relacionados ao tempo de renderização em nanosegundos para o *Nexus 4*

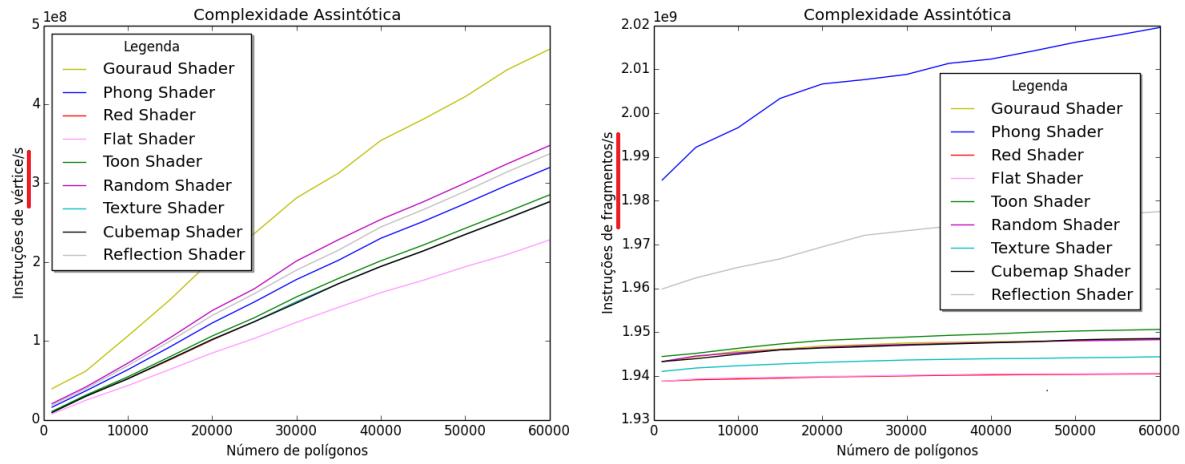


Figura 31 – Comparações entre as curvas dos *shaders*: todo processo de renderização

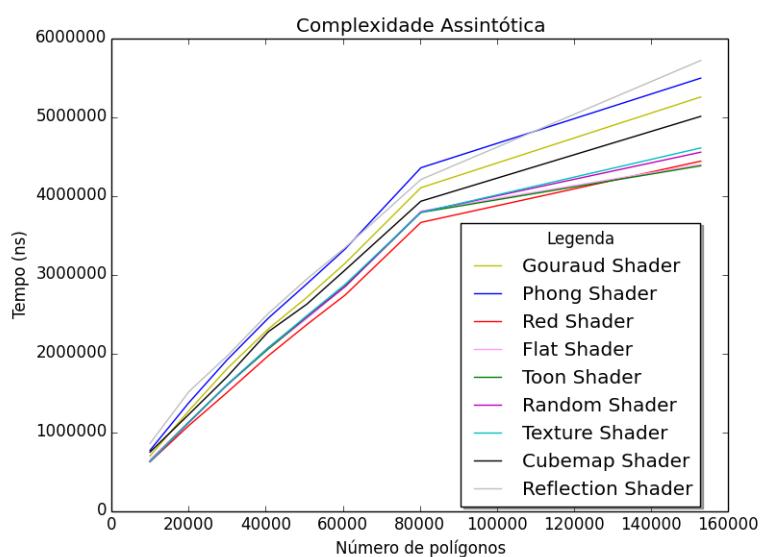


Figura 32 – Comparações entre as curvas dos *shaders*: *vertex* e *fragment shaders*

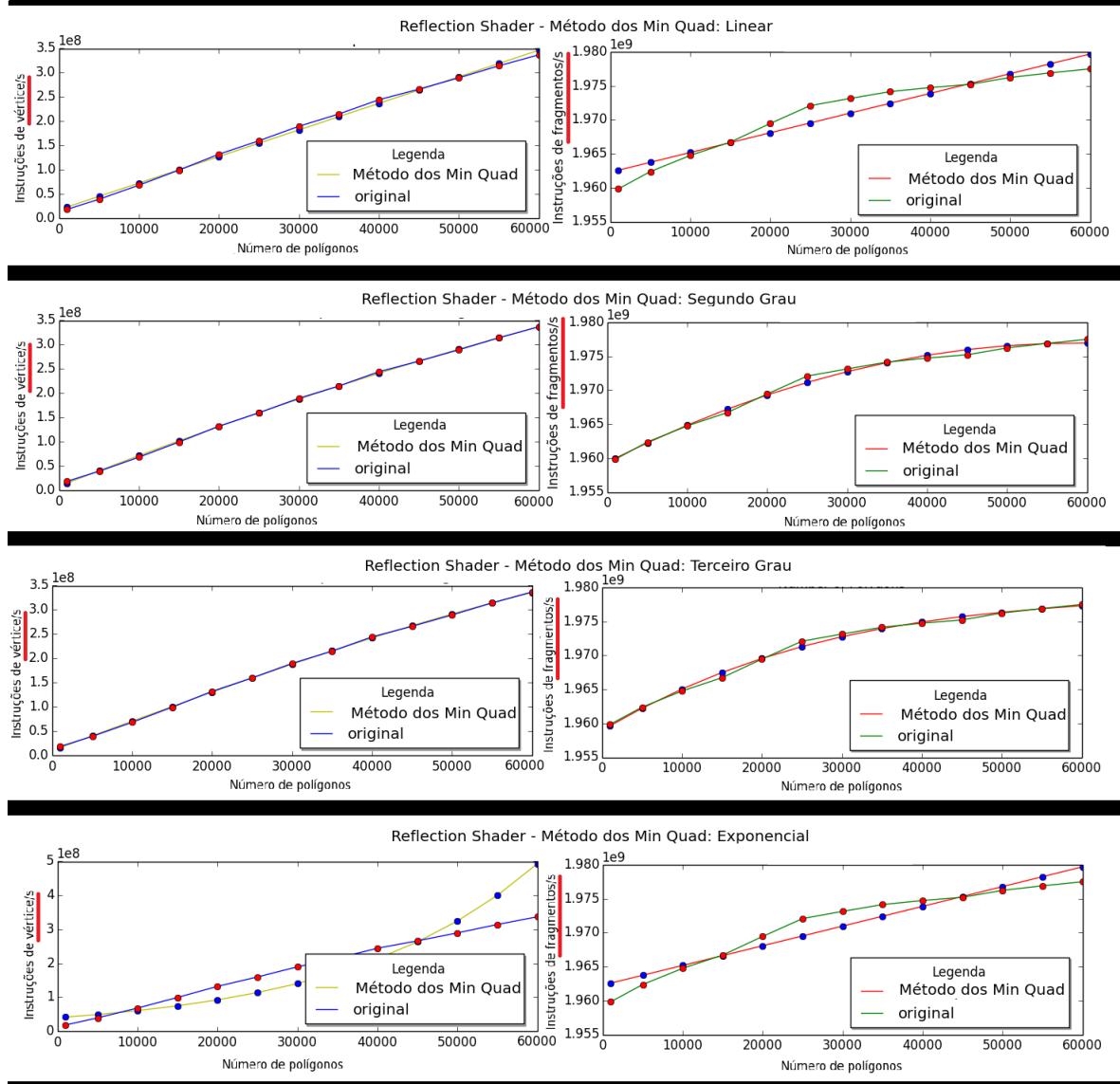


Figura 33 – Ajustes linear, segundo, terceiro graus e exponencial de cada tipo de *shader* para o *Nexus 4*

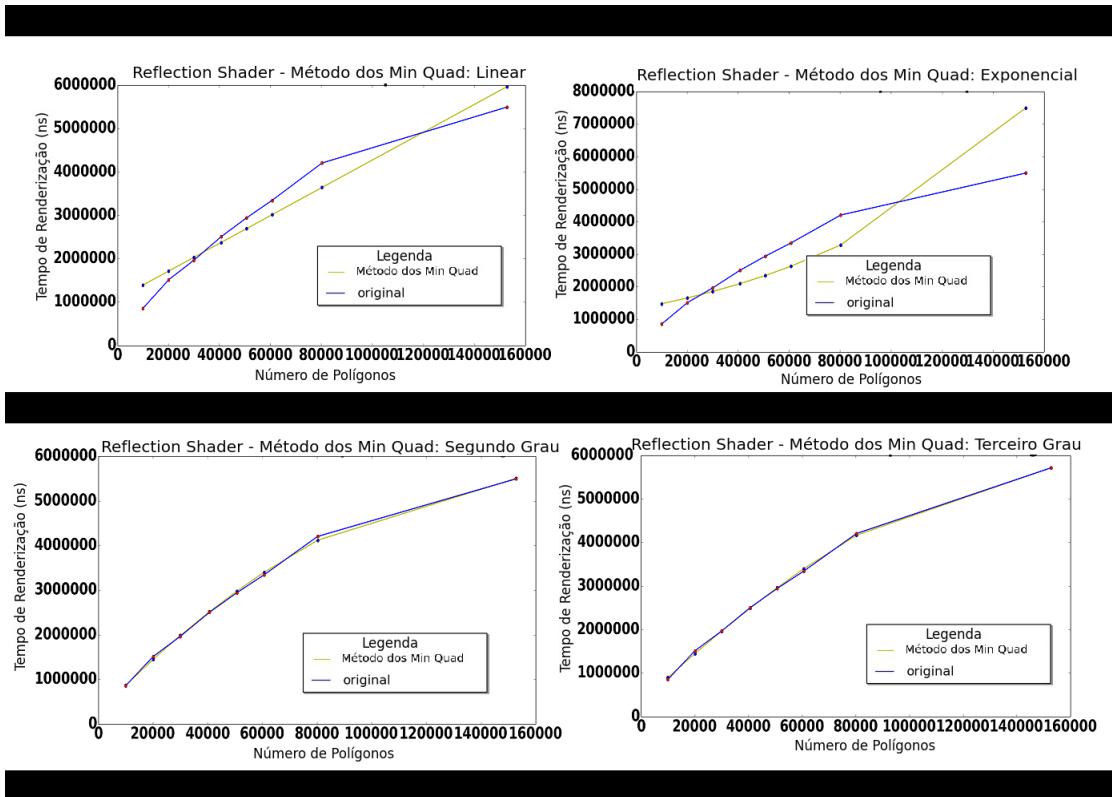


Figura 34 – Ajustes linear, segundo, terceiro graus e exponencial do processo de renderização para o *Nexus 4*

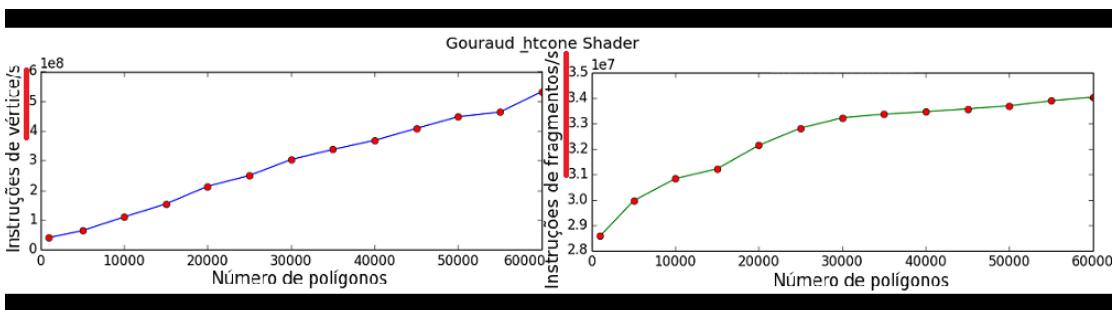


Figura 35 – Gráficos do *Gouraud Shader* para o *HTC One*

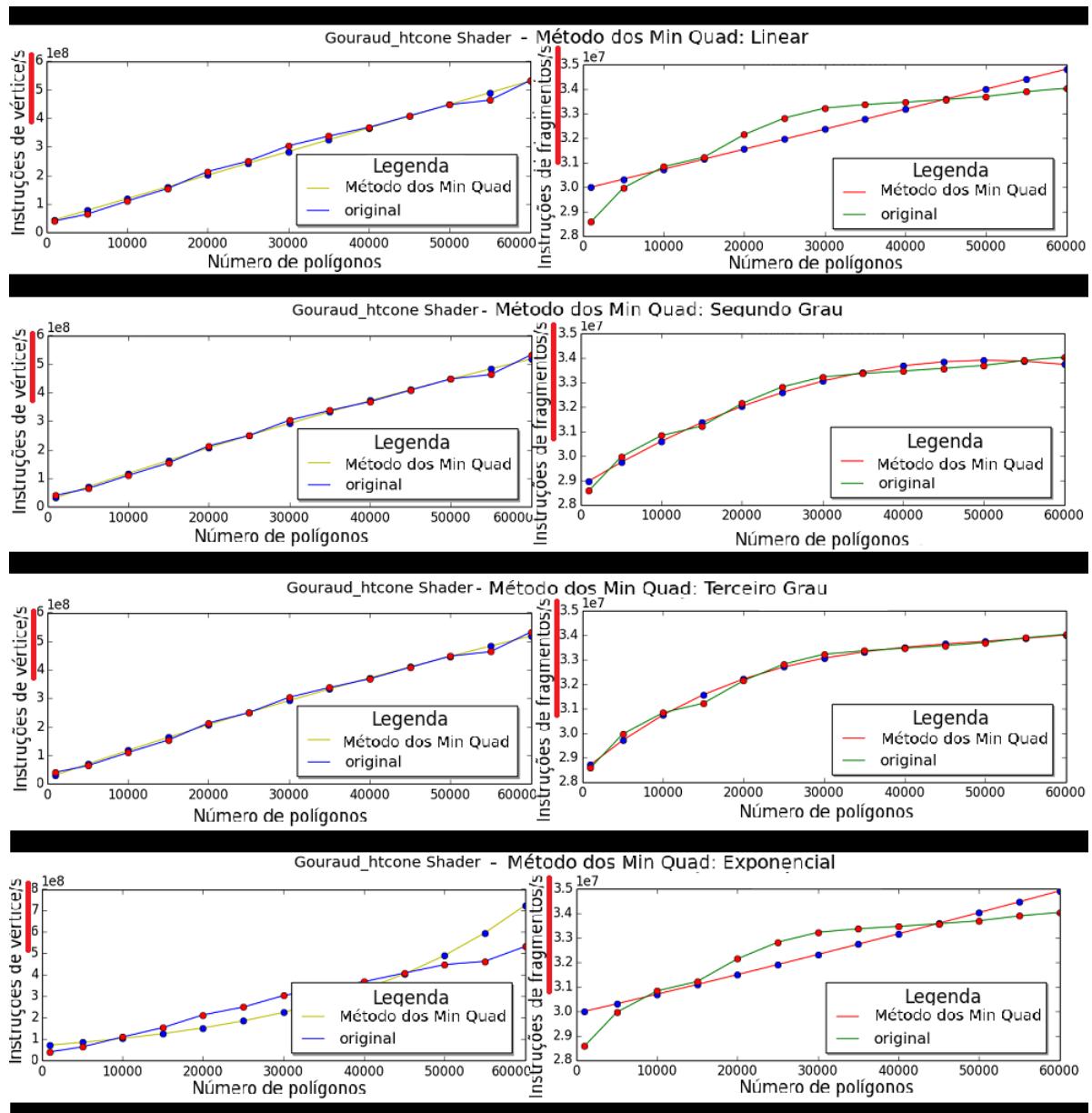


Figura 36 – Ajustes linear, segundo, terceiro graus e exponencial de cada tipo de *shader* para o *HTC One*

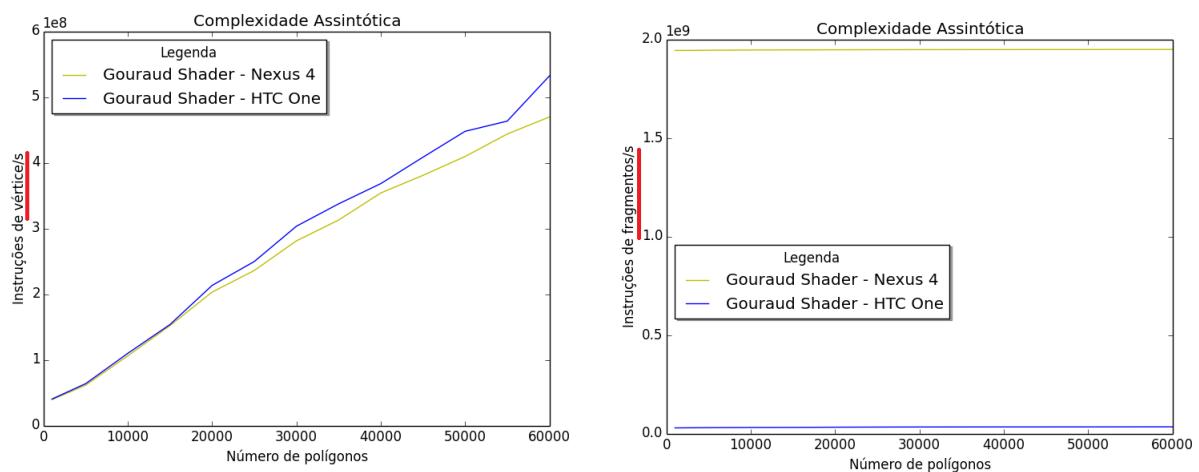


Figura 37 – Comparação entre as curvas dos *shaders*: *Nexus 4* e *HTC One*

4.2 Dispositivos iOS

Para os dispositivos *iPhone 5s* e *iPad Air*, foi possível apenas realizar as medições quanto ao processo de renderização, pois a ferramenta utilizada não provê medições quanto ao *vertex* e *fragment shaders*. A Figura 38 mostra o gráfico plotado para o *shader* tomado como exemplo de comparação, o *Gouraud Shader*. O formato das curvas obtidas são semelhantes ao do *Nexus 4*, e assim como ele, o melhor ajuste foi uma curva de terceiro grau (Figura 39).

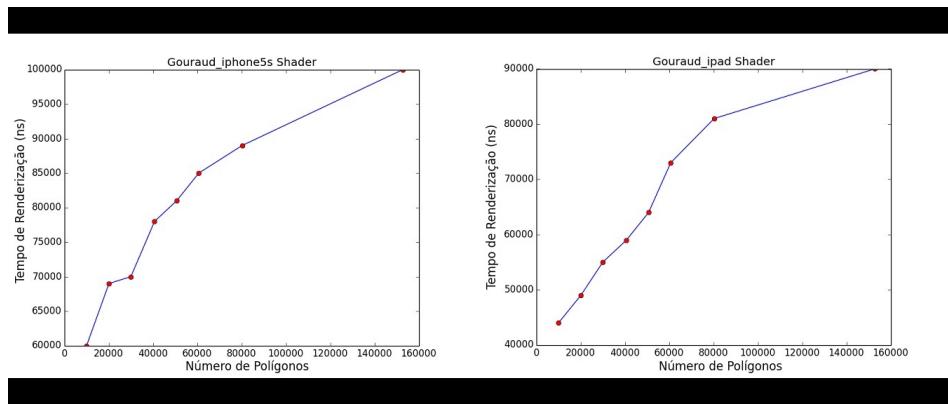


Figura 38 – Gráficos relacionados ao tempo de renderização em nanosegundos para os dispositivos *iOS*

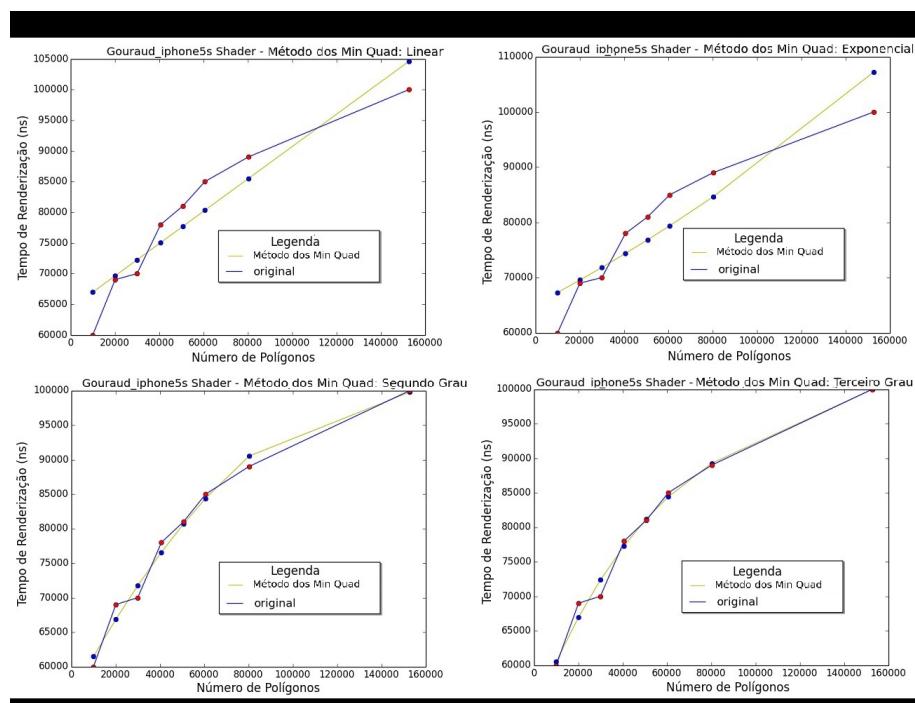


Figura 39 – Ajustes linear, segundo, terceiro graus e exponencial do processo de renderização para dispositivo *iOS*

A Figura 40 faz uma comparação entre as curvas dos *shaders* com relação ao *Nexus 4* e os dispositivos da plataforma *iOS* que são plotadas em um mesmo gráfico.

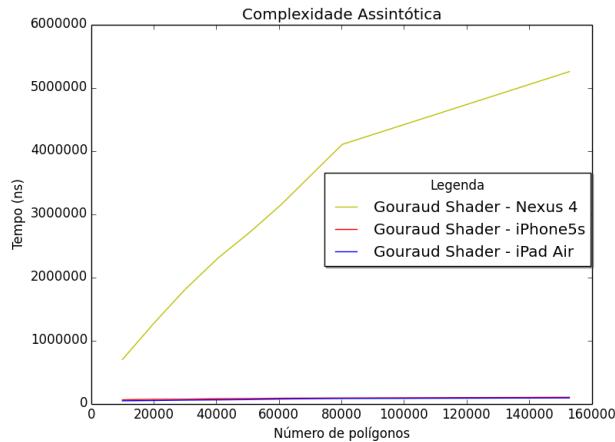


Figura 40 – Comparação entre as curvas dos *shaders*: *Nexus 4*, *iPhone 5s* e *iPad Air*

4.3 Análise das Equações Obtidas

As equações calculadas para cada *shader* do dispositivo *Nexus 4* (relacionadas ao vértice e fragmento) são mostradas na Tabela 8. Embora as curvas sejam de mesma família, os seus coeficientes não são idênticos. Os *shaders* relativamente mais simples tem inclinação de reta menor, assim como os coeficientes dos termos x^2 e x^3 . 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*, cuja curva aproximada para o *vertex shader* é $y(t) = 10.17 \times 10^6 + 4673.96t$, 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 (28.49×10^7).

As equações relacionadas a todo o processo de renderização do dispositivo *Nexus 4* também foram calculadas e podem ser vistas na Tabela 9. 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 estavam entre os de pior desempenho.

Outro resultado relevante está relacionado ao *Gouraud* e ao *Phong Shader*, 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 assintótica $O(n^3)$ e o *vertex shader*, $O(n)$, influenciando neste pior desempenho.

Pelo experimento realizado, os *shaders* de melhores desempenhos foram o *Flat*, *Toon* e *Red*. Além disso, utilizando as equações calculadas, a fim de estimar o tempo para 170.000 polígonos, por exemplo, essa análise se confirma, como é mostrada na Tabela 10.

Nome	Instruções por Segundo por Vértice	Instruções por Segundo por Fragmento
<i>Gouraud</i>	$y = 40,16 \times 10^6 + 7486,43t$	$y = 19,43 \times 10^8 + 297,00t - 0,0065t^2 + 0,50 \times 10^{-7}t^3$
<i>Phong</i>	$y = 14,95 \times 10^6 + 5211,02t$	$y = 19,84 \times 10^8 + 1752,43t - 0,0389t^2 + 3,32 \times 10^{-7}t^3$
<i>Red</i>	$y = 8,02 \times 10^6 + 4545,69t$	$y = 19,39 \times 10^8 + 64,34t - 0,00090t^2 + 0,05 \times 10^{-7}t^3$
<i>Toon</i>	$y = 10,17 \times 10^6 + 4673,96t$	$y = 19,44 \times 10^8 + 268,89t - 0,0044t^2 + 0,30 \times 10^{-7}t^3$
<i>Flat</i>	$y = 7,65 \times 10^6 + 3738,61t$	$y = 19,39 \times 10^8 + 74,94t - 0,0013t^2 + 0,08 \times 10^{-7}t^3$
<i>Random Color</i>	$y = 20,58 \times 10^6 + 5640,13t$	$y = 19,43 \times 10^8 + 250,33t - 0,0050t^2 + 0,37 \times 10^{-7}t^3$
<i>Simple Texture</i>	$y = 8,80 \times 10^6 + 4540,32t$	$y = 19,41 \times 10^8 + 160,00t - 0,0030t^2 + 0,22 \times 10^{-7}t^3$
<i>CubeMap</i>	$y = 8,67 \times 10^6 + 4540,40t$	$y = 19,43 \times 10^8 + 245,89t - 0,0047t^2 + 0,37 \times 10^{-7}t^3$
<i>Reflection</i>	$y = 18,03 \times 10^6 + 5470,95t$	$y = 19,59 \times 10^8 + 698,57t - 0,0094t^2 + 0,47 \times 10^{-7}t^3$

Tabela 8 – Equações relacionadas ao *vertex shader* e *fragment shader*

4.3.1 Comparação das Equações dos Dispositivos

A equação do processo de renderização também foi obtida para os dispositivos *iOS*, a partir de um *shader* de comparação (*Gouraud Shader*). A Tabela 11 mostra a comparação entre as equações destes dispositivos e o *Nexus 4*. Pelas medições realizadas

Nome	Tempo do Processo de Renderização (ns)
Gouraud	$y = 24,31 \times 10^4 + 48,89t + 7,60 \times 10^{-5}t^2 - 1,19 \times 10^{-9}t^3$
Phong	$y = 31,25 \times 10^4 + 49,28t + 0,12 \times 10^{-5}t^2 - 1,43 \times 10^{-9}t^3$
Red	$y = 30,37 \times 10^4 + 32,92t + 0,26 \times 10^{-5}t^2 - 0.00019 \times 10^{-9}t^3$
Toon	$y = 27,28 \times 10^4 + 37,30t + 0,23 \times 10^{-5}t^2 - 1,93 \times 10^{-9}t^3$
Flat	$y = 32,82 \times 10^4 + 33,84t + 0,28 \times 10^{-5}t^2 - 2,15 \times 10^{-9}t^3$
Random Color	$y = 26,25 \times 10^4 + 38,42t + 0,20 \times 10^{-5}t^2 - 1,76 \times 10^{-9}t^3$
Simple Texture	$y = 24,51 \times 10^4 + 38,88t + 0,18 \times 10^{-5}t^2 - 1,65 \times 10^{-9}t^3$
CubeMap	$y = 29,87 \times 10^4 + 44,70t + 0,11 \times 10^{-5}t^2 - 1,28 \times 10^{-9}t^3$
Reflection	$y = 33,63 \times 10^4 + 57,31t - 9,18 \times 10^{-5}t^2 - 0,35 \times 10^{-9}t^3$

Tabela 9 – Equações relacionadas ao tempo do processo de renderização

Nome	Tempo do Processo de Renderização (ns)
Gouraud	4.923.257
Phong	5.072.434
Red	3.825.594
Toon	3.687.182
Flat	3.653.955
Random Color	3.967.677
Simple Texture	4.065.117
CubeMap	4.638.068
Reflection	5.727.936

Tabela 10 – Exemplo de estimativa para 170.000 polígonos

e equações obtidas, foi possível perceber que o dispositivo de melhor desempenho foi o *iPad Air*, que como é mostrado na Seção 3.2, é o dispositivo com a melhor configuração de *hardware* e o que obteve melhor posição no aplicativo de *benchmarking* também mostrado nesta seção. O de pior desempenho foi o *Nexus 4*, que dentre estes dispositivos é o que tem a pior configuração de *hardware*.

Dispositivo	Tempo do Processo de Renderização (ns)
<i>Nexus 4</i>	$y = 24,31 \times 10^4 + 48,89t + 7,60 \times 10^{-5}t^2 - 1,19 \times 10^{-9}t^3$
<i>iPhone 5s</i>	$y = 5,32 \times 10^4 + 0,79t - 0,54 \times 10^{-5}t^2 + 0,02 \times 10^{-9}t^3$
<i>iPad Air</i>	$y = 4,03 \times 10^4 + 0,35t + 0,44 \times 10^{-5}t^2 - 0,03 \times 10^{-9}t^3$

Tabela 11 – Equações do processo de renderização do *Gouraud Shader*

As equações quanto ao *vertex* e *fragment shaders* também foram obtidas para o dispositivo *HTC One*. A Tabela 12 mostra a comparação entre as equações deste dispositivo e o *Nexus 4*. Pelas medições realizadas e equações obtidas, o *HTC One* teve melhor desempenho com relação ao *fragment shader*. Pela Seção 3.2 ele é um dispositivo superior ao *Nexus 4* e obteve posição um pouco melhor no aplicativo de *benchmarking*. E isto

é confirmado pelo resultado obtido, pois o processo de fragmento possui complexidade assintótica maior que o de vértice.

Dispositivo	Instruções por Segundo por Vértice	Instruções por Segundo por Fragmento
<i>Nexus 4</i>	$y = 40,16 \times 10^6 + 7486,43t$	$y = 19,43 \times 10^8 + 297,00t - 0,0065t^2 + 5,06 \times 10^{-8}t^3$
<i>HTC One</i>	$y = 357,56 \times 10^6 + 8251,00t$	$y = 0,29 \times 10^8 + 279,63t - 0,0052t^2 + 3,55 \times 10^{-8}t^3$

Tabela 12 – Equações do *vertex shader* e *fragment shaders* para o *Gouraud Shader*

4.3.2 Considerações Finais das Equações

Por meio dos resultados, foi possível perceber que o processo de renderização como um todo, calculado por meio do tempo de renderização da GPU e pela análise dos erros quadráticos, e o processo relacionado ao *fragment shader* tenderam a apresentar como complexidade assintótica um polinômio de terceiro grau para qualquer *shader* (variando somente os coeficientes das funções).

Porém, mesmo que os erros quadráticos sejam menores para as curvas de terceiro grau (relacionadas a todo o processo de renderização e ao *fragment shader*), os coeficientes das equações mostradas na Seção 4.3 para o termo t^3 são muito pequenos, sendo da ordem de 10^{-7} , 10^{-8} e 10^{-9} . No caso da ordem de 10^{-7} , por exemplo, será somado/subtraído (de acordo com o sinal), uma unidade a cada 100 milhões de unidades de t (para uma função $y(t)$), o que pode ser considerado irrelevante. Assim, neste contexto, a curva relacionada ao polinômio de segundo grau, mesmo com um erro quadrático maior, representa melhor a realidade do *shader*, em que pode-se considerar a complexidade assintótica do processo de renderização e do relacionado ao *fragment shader* $O(n^2)$. A Tabela 13 e a Tabela 14 mostram as equações obtidas para o polinômio de segundo grau, relacionadas ao *fragment shader* e a todo o processo de renderização, respectivamente, para o dispositivo *Nexus 4*. As análises feitas na Seção 4.3 para os polinômios de terceiro grau também se aplicam para os de segundo grau, em que o *shader* de pior desempenho (relacionado ao processo do *fragment shader*) é o *Phong Shader*, por exemplo.

A Tabela 15 e a Tabela 16 mostram as equações atualizadas para os polinômios de segundo grau para as comparações entre os dispositivos. Assim, percebe-se que os resultados das comparações feitas na Seção 4.3 também continuam os mesmos, ainda que as equações tenham sido mudadas do polinômio de terceiro grau para o de segundo grau.

Nome	Instruções por Segundo por Fragmento
Gouraud	$y = 19,43 \times 10^8 + 187,41t - 0,0019t^2$
Phong	$y = 19,87 \times 10^8 + 1034,36t - 0,0087t^2$
Red	$y = 19,39 \times 10^8 + 53,58t - 0,00044t^2$
Toon	$y = 19,44 \times 10^8 + 204,84t - 0,0017t^2$
Flat	$y = 19,39 \times 10^8 + 57,12t - 0,00050t^2$
Random Color	$y = 19,44 \times 10^8 + 170,31t - 0,0016t^2$
Simple Texture	$y = 19,41 \times 10^8 + 112,05t - 0,0010t^2$
CubeMap	$y = 19,43 \times 10^8 + 165,99t - 0,0014t^2$
Reflection	$y = 19,59 \times 10^8 + 596,55t - 0,0051t^2$

Tabela 13 – Polinômios de segundo grau relacionados ao *fragment shader*

Nome	Tempo do Processo de Renderização (ns)
Gouraud	$y = 3,64 \times 10^4 + 64,62t - 0,00020t^2$
Phong	$y = 6,263 \times 10^4 + 68,29t - 0,00021t^2$
Red	$y = -3,64 \times 10^4 + 58,80t - 0,00019t^2$
Toon	$y = -6,36 \times 10^4 + 62,91t - 0,00022t^2$
Flat	$y = -4,58 \times 10^4 + 62,31t - 0,00022t^2$
Random Color	$y = -4,37 \times 10^4 + 61,74t - 0,00021t^2$
Simple Texture	$y = -4,18 \times 10^4 + 61,72t - 0,00020t^2$
CubeMap	$y = 7,60 \times 10^4 + 61,64t - 0,00019t^2$
Reflection	$y = 27,57 \times 10^4 + 61,93t - 0,00017t^2$

Tabela 14 – Polinômios de segundo grau relacionados ao tempo do processo de renderização

Dispositivo	Tempo do Processo de Renderização (ns)
Nexus 4	$y = 3,64 \times 10^4 + 64,62t - 198,66 \times 10^{-6}t^2$
iPhone 5s	$y = 5,57 \times 10^4 + 0,59t - 2,00 \times 10^{-6}t^2$
iPad Air	$y = 3,51 \times 10^4 + 0,75t - 2,53 \times 10^{-6}t^2$

Tabela 15 – Equações do processo de renderização do *Gouraud Shader*

Dispositivo	Instruções por Segundo por Vértice	Instruções por Segundo por Fragmento
Nexus 4	$y = 40,16 \times 10^6 + 7486,43t$	$y = 19,43 \times 10^8 + 187,41t - 0,0019t^2$
HTC One	$y = 357,56 \times 10^6 + 8251,00t$	$y = 28,77 \times 10^6 + 202,81t - 0,0020t^2$

Tabela 16 – Equações do *vertex* e *fragment shaders* para o *Gouraud Shader*

4.4 Estimativas em Ambientes de Produção

No contexto de desenvolvimento de jogos, a unidade comumente utilizada para determinar o desempenho médio de uma aplicação é o FPS (*Frames Per Second*) ou quadros por segundo, que é o número de imagens renderizadas por segundo. Assim, também é possível converter os resultados obtidos para esta unidade, por meio da Equação 4.1, onde t é o tempo em segundos (obtido do processo de renderização). Também é possível obter este tempo por meio da medição do número de instruções por segundo, como mostra a Equação 4.2, pois a ferramenta *Adreno Profiler* também informa a quantidade de instruções para um *frame*.

$$FPS = \frac{1}{t} \quad (4.1)$$

$$t = \frac{I_F}{I_{PS}} \quad (4.2)$$

onde I_F é o número de instruções para um *frame* e I_{PS} é o número de instruções por segundo.

Além disso, antes da conversão em *frames* por segundo, somou-se à variável t os tempos das outras funções da *OpenGL ES* utilizadas para um *frame* (que não variam com a quantidade de polígonos). Para a plataforma *Android* estes tempos foram obtidos por meio da extensão da *OpenGL ES* utilizada. Já na plataforma *iOS*, a ferramenta *Instruments* informa o tempo de cada função utilizada.

Assim, a Tabela 17 mostra os resultados convertidos em *frames* por segundo, para o *shader Gouraud* utilizando o equipamento *Nexus 4*, fazendo a ressalva de que esta medição não inclui outros fatores presentes num ambiente real de produção, como *input* e física, por exemplo.

Quantidade de Polígonos	Segundos Renderização	- Segundos - Outras Funções	FPS
10.000	0,000698	0,0000140	1405
20.100	0,00127	0,0000140	779
30.000	0,00181	0,0000140	548
40.678	0,00231	0,0000140	430
50.679	0,00271	0,0000140	367
60.662	0,00315	0,0000140	316
80.256	0,00410	0,0000140	243
152.840	0,00525	0,0000140	190

Tabela 17 – Estimativa de FPS

4.5 Processo Experimental da Estimação da Complexidade Assintótica

Assim, o processo utilizado neste trabalho de estimação da complexidade algorítmica calculada de forma empírica, pode ser resumido na Figura 41. A etapa de Implementar *Shaders* pode ser feita por meio da utilização da base do projeto implementado, herdando-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.

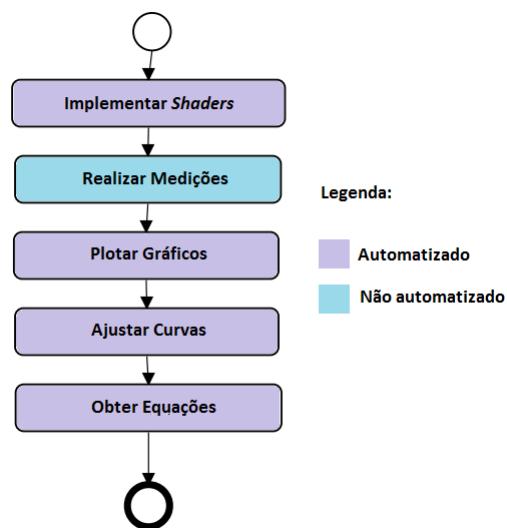


Figura 41 – Processo da Análise de Complexidade Algorítmica.

5 Conclusão

Por meio dos experimentos realizados, foi possível perceber que a complexidade assintótica, no que diz respeito ao processo relacionado ao *vertex shader*, se comportou linearmente, independentemente do *shader* utilizado. Assim, todos os *shaders* avaliados possuem a mesma complexidade algorítmica, mas as equações de cada um possuem coeficientes diferentes, que podem determinar qual *shader* tem melhor ou pior desempenho. Analisando a teoria do processo de renderização da *OpenGL* para o *vertex shader*, este resultado é consistente, pois o programa do *vertex shader* é utilizado para cada vértice, dependendo então do número de vértices e da complexidade do *shader*. Assim, o fluxo de execução da aplicação do *vertex shader* pode ser representado de acordo com o Código 5.1.

Código 5.1 – Representação da aplicação do *vertex shader*

```
1 for(int i = 0; i < vertexBuffer.length; i++)
2 {
3     executeVertexShader(vertexBuffer[i]);
}
```

Já o processo de renderização como um todo e o relacionado ao *fragment shader* tenderam a apresentar como complexidade assintótica um polinômio de segundo grau. O Código 5.2 mostra um fluxo genérico de execução da aplicação do *fragment shader*. Assim, como é explicado na documentação da *OpenGL*¹ e no Anexo A, para cada primitiva da malha (no caso os triângulos), geram-se os fragmentos (possíveis candidatos a *pixel*). E para cada fragmento, faz-se o processo de aplicação do *fragment shader*, ressaltando que para cada fragmento percorre-se a direção horizontal e vertical da tela (sendo uma matriz). Assim, a função `executeFragmentShader(fragment)`, atribui a um fragmento uma cor e um valor de profundidade (este valor é usado nos passos sequentes do processo de renderização para descarte de fragmentos). Então possivelmente a complexidade assintótica quadrática está associada à atribuição da cor (que percorre uma matriz, sendo de ordem quadrada).

Código 5.2 – Representação da aplicação do *fragment shader*

```
1 triangleStream = Mesh.triangles;
2
3 for(int i = 0; i < triangleStream.length; i++)
4 {
5     fragmentStream = triangleStream[i].fragments;
6 }
```

¹ http://www.opengl.org/wiki/Fragment_Shader

```

7     for(int j = 0; j < fragmentStream.length;j++)
8     {
9         executeFragmentShader(fragmentStream[i]);
10    }
11
12 }
```

Além disso, os resultados obtidos não são tão óbvios, pois ao analisar o programa do *vertex shader* do Código 5.3, por exemplo, há somente uma atribuição, induzindo o programador a achar que a complexidade é constante, em que o trabalho desenvolvido evidenciou a verdadeira complexidade do algoritmo.

Código 5.3 – 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 assintótica, uma forma de comparar o desempenho entre eles, para um mesmo dispositivo, 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 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).

Além disso, também é possível comparar, para um mesmo *shader*, diferentes dispositivos por meio das equações e valores das medições obtidos. As comparações realizadas neste trabalho foram condizentes com a expectativa, o dispositivo *iPhone 5s*, que é de uma geração de *smartphones* mais recente, por exemplo, teve desempenho muito melhor que o *Nexus 4*.

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 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 85.
- BUCK, E. *Learning OpenGL ES for iOS*: A hands-on guide to modern 3d graphics programming. 1. ed. Crawfordsville, Indiana: Pearson, 2012. Citado na página 25.
- DROZDEK, A. *Estrutura de Dados e Algoritmos em C++*. 2. ed. São Paulo, São Paulo: Cengage Learning, 2002. Citado na página 29.
- 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 27.
- GUHA, S. *Computer Graphics Through OpenGL*: From theory to experience. 1. ed. Boca Raton, Florida: CRC Press, 2011. Citado 2 vezes nas páginas 25 e 26.
- JACKSON, W. *Learn Android App Development*. 1. ed. New York, New York: Apress, 2013. Citado na página 41.
- LEITHOLD, L. *O Cálculo com Geometria Analítica*. 3. ed. São Paulo, São Paulo: Harbra, 1994. Citado na página 31.
- 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.
- NEUBURG, M. *iOS 7 Programming Fundamentals*: Objective-c, xcode, and cocoa basics. 1. ed. Sebastopol, California: O'Reilly, 2013. Citado 2 vezes nas páginas 36 e 57.
- 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 39.
- RORRES, A. *Álgebra Linear com Aplicações*. 8. ed. Porto Alegre, Rio Grande do Sul: Bookman, 2001. Citado na página 31.
- 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 [29](#).

SHERROD, A. *Game Graphics Programming*. 1. ed. Boston, Massachusetts: Course Technology, 2011. Citado 2 vezes nas páginas [17](#) e [89](#).

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 42, 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.

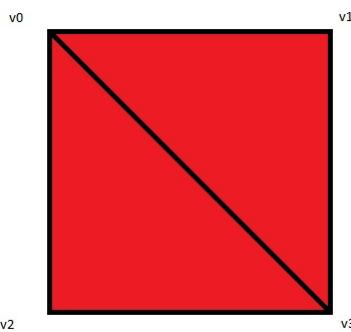


Figura 42 – Vértices do quadrado constituído de dois triângulos

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 43 ilustra esta ideia.

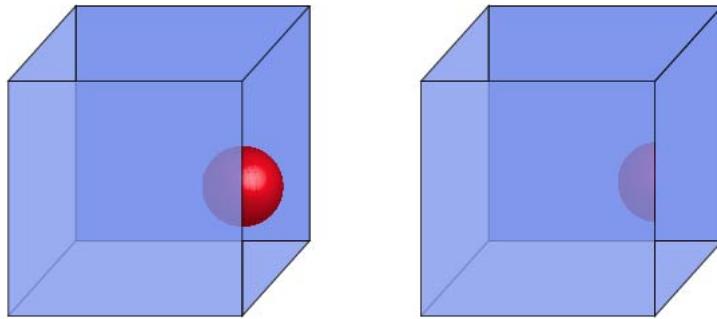


Figura 43 – Antes do recorte (cubo de visualização esquerdo) e depois do recorte (cubo de visualização direito)

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*).

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 44. 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.

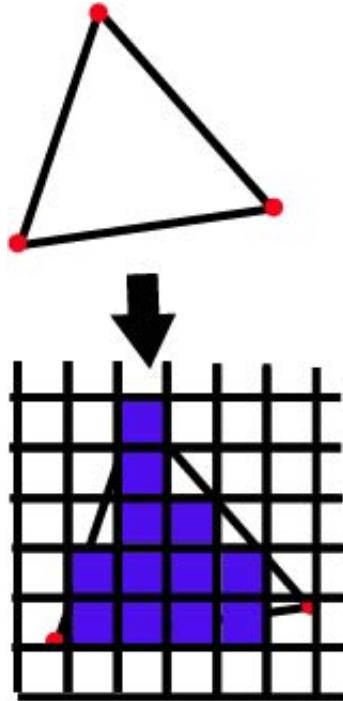


Figura 44 – Travessia de triângulos: fragmentos sendo gerados

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 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), em que 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 ([SHERROD, 2011](#)), neste arquivo, cada linha contém informações a respeito do modelo, começando com uma palavra-chave, seguida da informação. A Tabela 18 mostra as principais palavras-chave utilizadas.

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

Tabela 18 – 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.