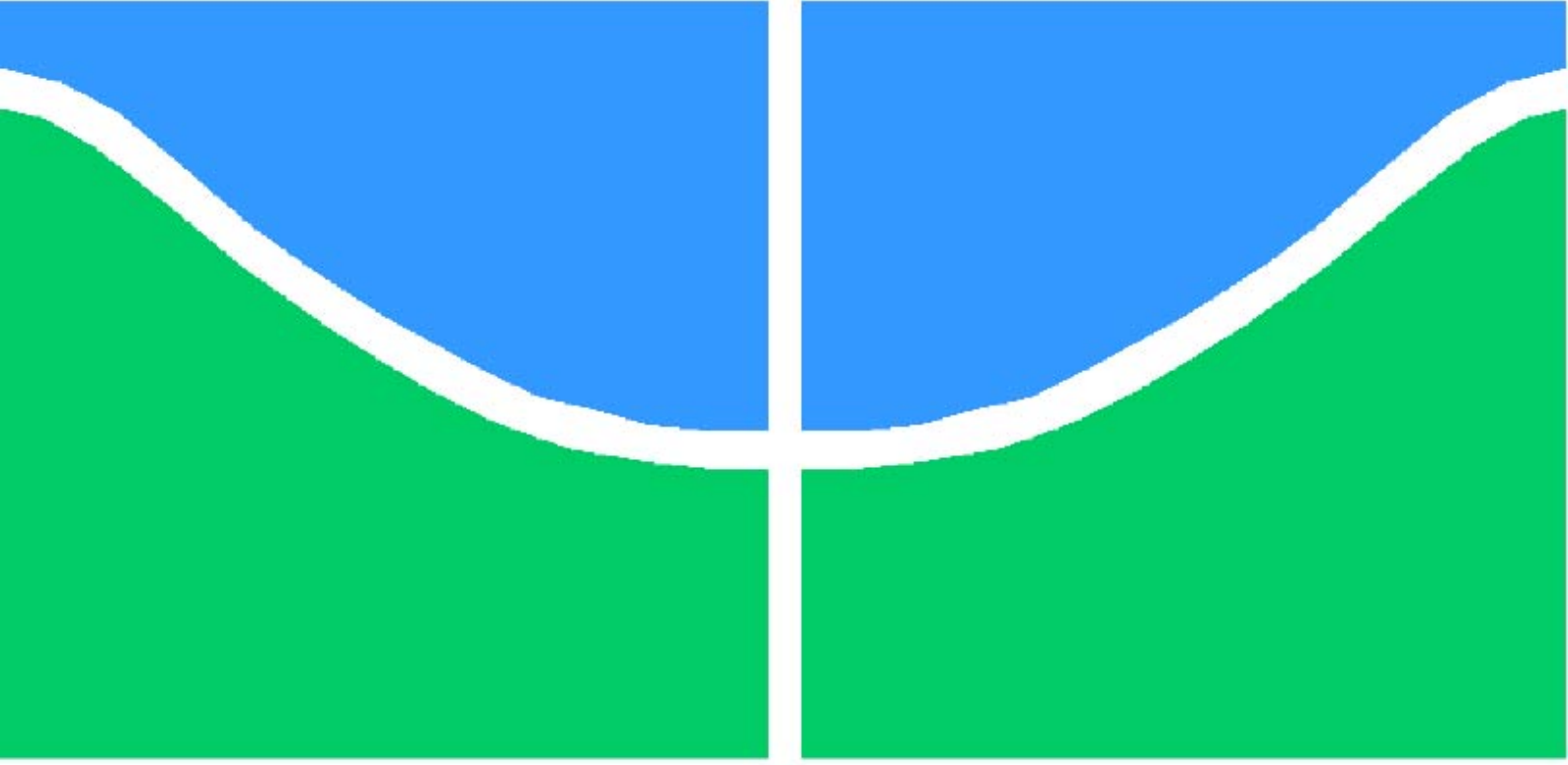

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de *Software*

# Implementação de *Shaders* para Plataforma *Android*

Autor: Aline de Souza Campelo Lima

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF

2013

Aline de Souza Campelo Lima

# Implementação de *Shaders* para Plataforma *Android*

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

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Brasília, DF

2013

Aline de Souza Campelo Lima
Implementação de *Shaders* para Plataforma *Android*/ Aline de Souza Campelo Lima. – Brasília, DF, 2013-
66 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Edson Alves da Costa Júnior

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2013.

1. Computação Gráfica. 2. Complexidade Algorítmica. I. Prof. Dr. Edson Alves da Costa Júnior. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Implementação de *Shaders* para Plataforma *Android*

CDU 02:141:005.6

Aline de Souza Campelo Lima

# Implementação de *Shaders* para Plataforma *Android*

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

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

---

**Prof. Dr. Edson Alves da Costa Júnior**
Orientador

---

**Prof. Dr. Ricardo Pezzoul Jacobi**
Convidado 1

---

**Prof. Dra. Carla Silva Rocha Aguiar**
Convidado 2

Brasília, DF
2013

# Resumo

A utilização dos dispositivos móveis e da plataforma *Android* tem crescido e constata-se a importância dos efeitos visuais em jogos, bem como a sua limitação atual de desempenho dos processadores gráficos destas plataformas. Assim, a proposta do trabalho se baseia no desenvolvimento de *shaders* (programas responsáveis pelos efeitos visuais) para a plataforma *Android* e para computador, em que suas complexidades algorítmicas serão analisadas, baseando-se na métrica de quadros por segundo em função do número de polígonos renderizados. Além disso, o método dos mínimos quadrados será utilizado para ajustar os valores obtidos a uma curva, permitindo assim a estimativa da quantidade máxima de polígonos para a execução de um programa na taxa de quadros por segundo desejada para um determinado *shader*.

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

# Abstract

The usage of mobile devices and Android platform is emerging and is notable the importance of visual effects in games and the performance restriction of the graphical processors of these platforms. This way, the purpose of this academic work is based on the development of shaders (programs responsible for the visual effects) for Android platform and computer, which algorithm complexities will be analyzed, based on the frames per second depending on the number of polygons rendered. Besides, the method of least squares will be used to adjust the values obtained from a curve, being able to estimate the maximum quantity of polygons to be executed for a specified frames per second rate, related to a specific shader.

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

# Lista de ilustrações

# Lista de tabelas

# Lista de abreviaturas e siglas

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

# Sumário

# 1 Introdução

## 1.1 Contextualização e Justificativa

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

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

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

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

## 1.2 Delimitação do Assunto

O tema consiste no desenvolvimento de *shaders* aplicados em objetos tridimensionais – com número de polígonos variável – que são utilizados na renderização de cenas, as quais permitem a coleta de medições quanto ao número de quadros por segundo. Desta forma, é possivel variar a quantidade de polígonos de um objeto e traçar um gráfico

quantidade de polígonos *versus* quadros por segundo utilizando um determinado *shader*. E assim, analisa-se experimentalmente a complexidade algorítmica desses *shaders*, para posterior aplicação do método dos mínimos quadrados (como explicado na Seção 2), a fim de estimar o número de quadros por segundo renderizados por um *shader* específico dado um número $n$ de polígonos, baseando-se na curva obtida experimentalmente pelos gráficos.

Como visto na Seção 2, a complexidade algorítmica não depende das condições do ambiente de realização dos experimentos. Um algoritmo possui a mesma complexidade mesmo sendo implementado utilizando-se diferentes linguagens de programação, por exemplo. Assim, é possivel aplicar a proposta em diferentes contextos, como utilizando os *shaders* em computador e em celulares.

A fim de analisar se o tema também podia ser expandido para o contexto *mobile* e verificar se o mesmo seria factível dentro do prazo estipulado, primeiramente desenvolveu-se um *shader* utilizando a técnica *Gouraud Shading* (Seção 2) aplicado em um octaedro (polígono regular de 8 faces), usando a linguagem Java (padrão do *Android*). O mesmo programa também foi implementado para computador utilizando a linguagem C++. Dessa forma, foi possível averiguar que o tema também poderia ser estendido e aplicado na plataforma *Android*, e evidenciou-se as principais diferenças entre a *OpenGL ES* – utilizada para celulares – e a *OpenGL* que é utilizada em computadores.

## 1.3 Objetivos Gerais

Os objetivos gerais do trabalho são a implementação de *shaders* na plataforma *Android* e no computador. Assim como a análise das suas complexidades algorítmicas, estimando a taxa de quadros por segundo para uma determinada quantidade de polígonos.

## 1.4 Objetivos Específicos

Os objetivos específicos do trabalho são:

- Analisar quais *shaders* serão implementados;
- Configurar os ambientes de desenvolvimento tanto para computador, como para a plataforma *Android*;
- Identificar qual métrica será utilizada para a análise de complexidade;
- Verificar se existem ferramentas que coletem a métrica definida;
- Configurar as ferramentas de coleta de medições, caso existam;

- Coletar as medições estabelecidas;
- Estimar a quantidade de polígonos renderizados de acordo com a quantidade de quadros por segundo desejada.

## 1.5 Organização do Trabalho

No próximo capítulo serão apresentados os conceitos teóricos necessários para o entendimento do trabalho, como, por exemplo, o processo de renderização, a biblioteca gráfica utilizada, definição da plataforma *Android*, definição da métrica quadros por segundo, complexidade algorítmica, entre outros.

Na metodologia, os passos tomados no trabalho são descritos, enfatizando como foi feito o levantamento bibliográfico e a configuração do ambiente, quais equipamentos foram utilizados, que abordagem foi utilizada para definir o tema e quais os próximos passos a serem tomados. Além disso, são descritas as decisões tomadas para evidenciar a viabilidade do trabalho.

Nos resultados alcançados são descritos os resultados preliminares da implementação no computador e na plataforma *Android*, seguido das conclusões que se seguirão aos trabalhos realizados.

# 2 Referencial Teórico

## 2.1 *Shaders*: *pipelines* programáveis

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

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

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

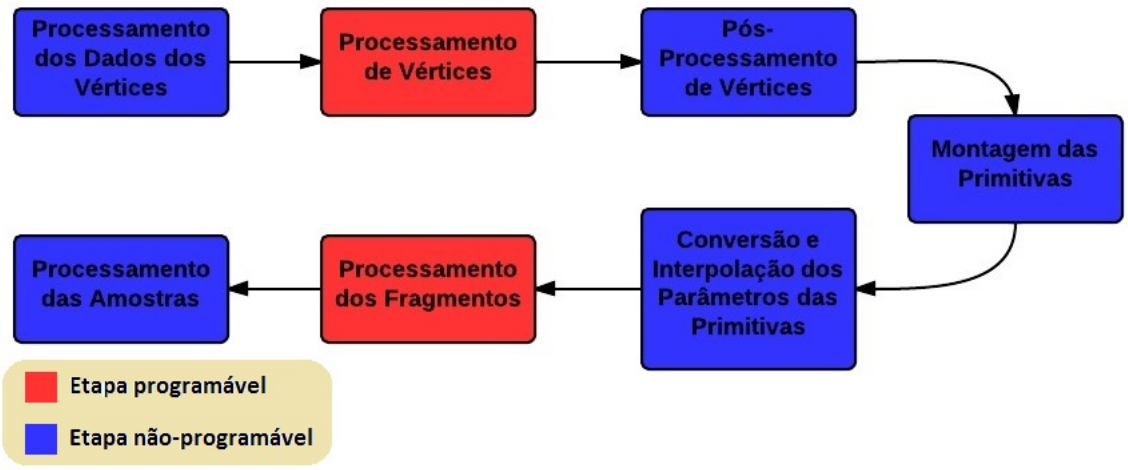


Figura 1 – Processo de renderização da *OpenGL*

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

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

### 2.1.1 Linguagens dos *shaders*

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

| **Linguagem de Programação** | **Biblioteca Gráfica Suportada** |
|---|---|
| *GLSL: OpenGL Shading Language* | *OpenGL* |
| *HLSL: High Level Shading Language* | *DirectX* e *XNA* |
| *Cg: C for Graphics* | *DirectX* e *OpenGL* |

Tabela 1 – Linguagens de programação para *shaders*

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

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

Tabela 2 – GLSL: tipos de dados

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

### 2.1.2 Utilização dos *shaders* em *OpenGL*

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

| Tipo | Descrição |
|---|---|
| `attribute` | Variável utilizada pelo programa para comunicar dados relacionados aos vértices para o *vertex shader* |
| `uniform` | Variável utilizada pelo programa para comunicar dados relacionados com as primitivas para ambos os *shaders* |
| `varying` | Variável utilizada pelo *vertex shader* para se comunicar com o *fragment shader* |

Tabela 3 – GLSL: qualificadores

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

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

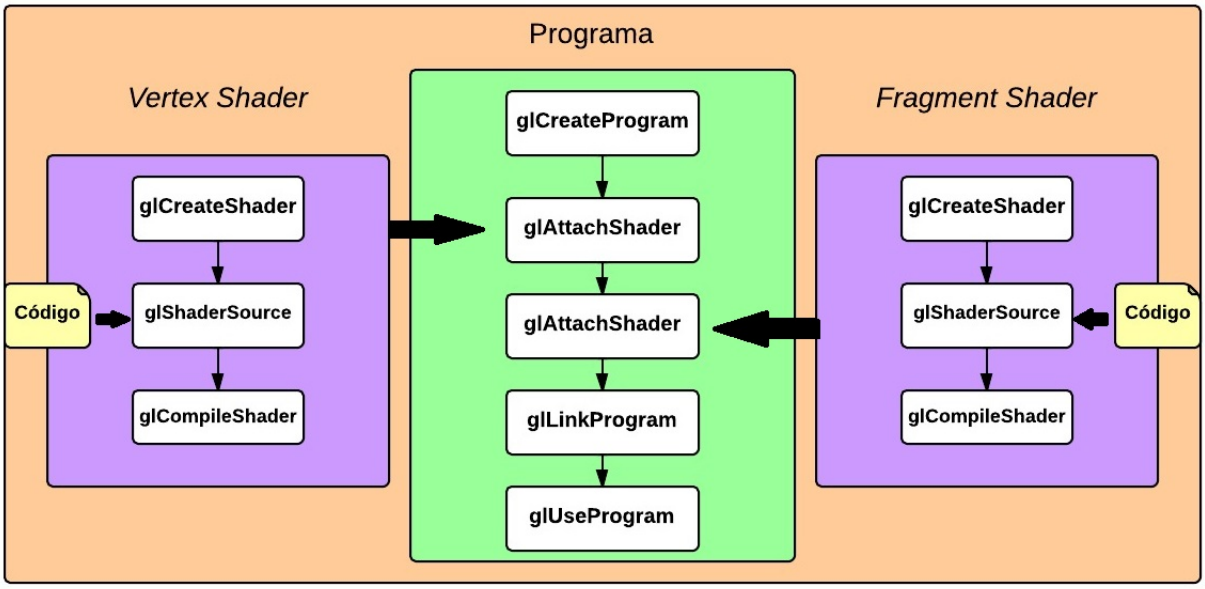


Figura 2 – Utilização dos *shaders*

### 2.1.3 *Shaders* em Plataformas Móveis

#### 2.1.3.1 Plataforma *Android*

O *Android* começou a ser desenvolvido em 2003 na empresa de mesmo nome, fundada por Andy Rubin, a qual foi adquirida em 2005 pela empresa *Google*. A *Google* criou a *Open Handset Alliance*, que junta várias empresas da indústria das telecomunicações, como a *Motorola* e a *Samsung*, por exemplo. Assim, elas desenvolveram o *Android* como é conhecido hoje, o qual é um sistema operacional *open source* para dispositivos móveis

(baseado no *kernel* do *Linux*), tendo a primeira versão beta lançada em 2007 e segundo (SANDBERG; ROLLINS, 2013), hoje é o sistema operacional para *mobile* mais utilizado.

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

| **Número da versão** | **Nome** |
|---|---|
| 1.5 | *Cupcake* |
| 1.6 | *Donut* |
| 2.0/2.1 | *Éclair* |
| 2.2 | *FroYo* |
| 2.3 | *Gingerbread* |
| 3.0/3.1/3.2 | *HoneyComb* |
| 4.0 | *Ice Cream Sandwich* |
| 4.1/4.2/4.3 | *Jelly Bean* |
| 4.4 | *KitKat* |

Tabela 4 – Versões da plataforma *Android*

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

#### 2.1.3.2 *OpenGL ES*

A *OpenGL ES* (*OpenGL for Embedded Systems*) foi lançada em 2003, sendo a versão da *OpenGL* para sistemas embarcados, e como citado em (GUHA, 2011), atualmente é uma das API's mais populares para programação de gráficos tridimensionais em

[1] http://www.eclipse.org/
[2] http://developer.android.com/sdk/index.html
[3] http://developer.android.com/tools/sdk/eclipse-adt.html
[4] http://developer.android.com/sdk/installing/studio.html

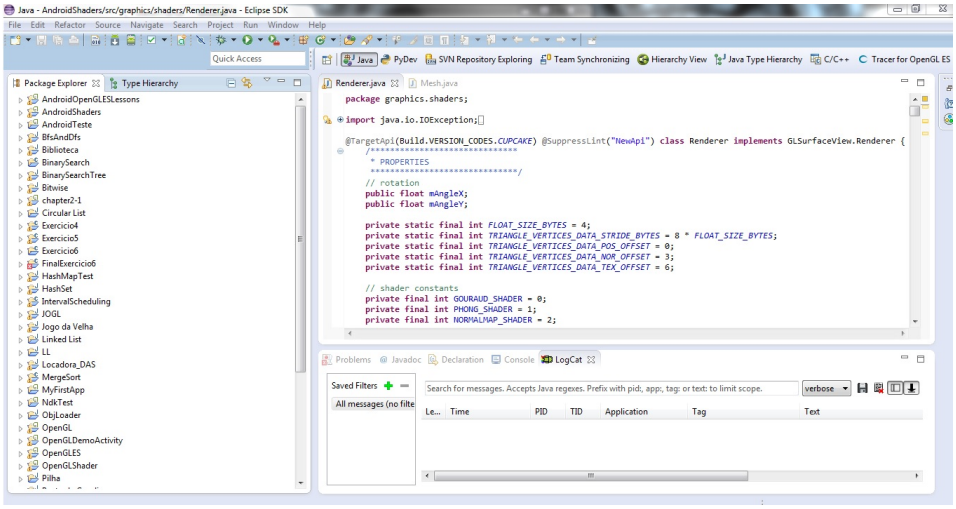


Figura 3 – Ambiente de desenvolvimento *Eclipse*

pequenos *devices*, sendo adotada por diversas plataformas como *Android*, *iOS*, Nintendo DS e *Black Berry*.

Segundo (SMITHWICK; VERMA, 2012), ela possui três versõe: 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.

## 2.2 Teoria Matemática para Implementação de *Shaders*

### 2.2.1 Equação de Iluminação de *Phong*

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

A computação dos cálculos de luz nos vértices seguida por uma interpolação linear do resultado é conhecida como *Gouraud Shading* (considerada superior ao *Flat Shading*, pois renderiza uma superfície mais suave, lisa), criada por Henri Gouraud, sendo conhecida como avaliação por vértice. Nela, o *vertex shader* deve calcular a intensidade em cada vértice e os resultados serão interpolados. Em seguida, o *fragment shader* pega este valor e passa adiante. Segundo (GUHA, 2011), é o padrão implementado pela *OpenGL*.

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 de luz é calculada de acordo com a equação de luz de *Phong* mostrada em (GUHA, 2011).

A *OpenGL* oferece este tipo de *shading* como opção, embora o mesmo possa ser implementado utilizando *shaders*. Ele 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 desvantagém da técnica de *Gouraud Shading* é que efeitos de luz que não afetam um vértice de uma superfície não surtirão efeito como, por exemplo, efeitos de luz localizados no meio de um polígono não serão renderizados corretamente. Porém, se o efeito ocorrer em um vértice, o *Phong Shading* renderiza corretamente o vértice, mas irá interpolar erroneamente. A Figura 4 mostra a diferença entre as três técnicas de *shading* aplicadas em uma esfera com uma luz direcional.

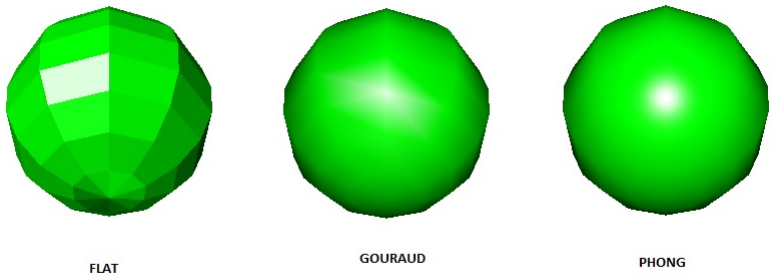


Figura 4 – Comparação entre as técnicas de *shading*

### 2.2.2 Renderização de Efeito *Cartoon*

### 2.2.3 Renderização de Efeito de Reflexão

## 2.3 Complexidade Algorítmica Calculada de Forma Empírica

### 2.3.1 Complexidade Algorítmica - Teoria

Complexidade algorítmica é uma medida que compara a eficiência de um determinado algorítmo, analisando o quão 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 algoritmica se baseia em uma função (medida lógica) que expressa uma relação entre a quantidade de dados e de tempo necessário para processá-los.

Como o cálculo visa a modelagem do comportamento do desempenho do algorítmo a medida que o número de dados aumenta, os termos que não afetam a ordem de

magnitude são eliminados, gerando a aproximação denominada complexidade assintótica. Assim, a Equação (2.1) poderia ser aproximada pela Equação (2.2)

$$y = n^2 + 10n + 1000 \tag{2.1}$$

$$y \approx n^2 \tag{2.2}$$

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

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

Tabela 5 – Valores mais comuns de complexidade algorítmica

### 2.3.2 Métodos dos Mínimos Quadrados

O método dos mínimos quadrados é utilizado para ajustar pontos $(x, y)$ determinados experimentalmente a uma curva. No caso do ajuste a uma reta (dada por $y = a + bx$) muitas vezes estes pontos não são colineares e segundo (RORRES, 2001) é impossível encontrar coeficientes $a$ e $b$ que satisfaçam o sistema.

Então, as distâncias destes valores para a reta podem ser consideradas como medidas de erro e os pontos são minimizados pelo mesmo vetor (minimizando a soma dos quadrados destes erros). Assim, existe um ajuste linear de mínimos quadrados aos dados,

e a sua solução é dada pela Equação (2.3), em que é possível determinar os coeficientes $a$ e $b$ e consequentemente, a equação da reta.

$$v = (M^T M)^{-1} M^T y \tag{2.3}$$

onde

$$M = \begin{bmatrix} 1 & x1 \\ 1 & x2 \\ . & . \\ . & . \\ 1 & xn \end{bmatrix}, \quad v = \begin{bmatrix} a \\ b \end{bmatrix} \text{ e } \quad y = \begin{bmatrix} y1 \\ y2 \\ . \\ . \\ yn \end{bmatrix} \tag{2.4}$$

De acordo com (LEITHOLD, 1994), a função da exponencial pode ser dada como na Equação (2.5), em que $e$, $c$, $k$ são constantes ($e$ é a constante neperiana).

$$y = ce^{-kt} \tag{2.5}$$

Aplicando a função logarítimo dos dois lados da equação, obtém-se a Equação (2.6)

$$\ln y = \ln c + \ln e^{-kt} \tag{2.6}$$

que pode ser simplificada na Equação (2.7) (em que $b$ é uma nova constante) que equivale à equação da reta.

$$\bar{y} = \bar{a} + \bar{b}t \tag{2.7}$$

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

$$M = \begin{bmatrix} 1 & x1 \\ 1 & x2 \\ . & . \\ . & . \\ 1 & xn \end{bmatrix}, \quad y = \begin{bmatrix} \ln y1 \\ \ln y2 \\ . \\ . \\ \ln yn \end{bmatrix} \tag{2.8}$$

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

$$c = e^{\bar{a}} \text{ e } \bar{b} = -k \tag{2.9}$$

## 2.4 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 6 mostra as principais palavras-chave utilizadas.

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

Tabela 6 – 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. A Figura 5 mostra o exemplo de um arquivo obj para a leitura de um cubo.

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.

```
#Formato OBJ - CUBO
v 2.000000 -2.000000 -2.000000
v 2.000000 -2.000000 2.000000
v -2.000000 -2.000000 2.000000
v -2.000000 -2.000000 -2.000000
v 2.000000 2.000000 -2.000000
v 2.000000 2.000000 2.000000
v -2.000000 2.000000 2.000000
v -2.000000 2.000000 -2.000000
#8 vértices
vt 0.000000 0.000000
vt 1.000000 0.000000
vt 1.000000 1.000000
vt 0.000000 1.000000
#4 coordenadas de textura
vn 0.578387 0.575213 -0.578387
vn 0.576281 -0.579455 -0.576281
vn -0.576250 -0.576281 -0.579455
vn -0.578387 0.578387 -0.575213
vn -0.577349 -0.577349 0.577349
vn -0.577349 0.577349 0.577349
vn 0.579455 -0.576281 0.576281
vn 0.575213 0.578387 0.578387
#6 normais
f 5/1/1 1/2/2 4/3/3
f 5/1/1 4/3/3 8/4/4
f 3/1/5 7/2/6 8/3/4
f 3/1/5 8/3/4 4/4/3
f 2/1/7 6/2/8 3/4/5
f 6/2/8 7/3/6 3/4/5
f 1/1/2 5/2/1 2/4/7
f 5/2/1 6/3/8 2/4/7
f 5/1/1 8/2/4 6/4/8
f 8/2/4 7/3/6 6/4/8
f 1/1/2 2/2/7 3/3/5
f 1/1/2 3/3/5 4/4/3
```

Figura 5 – Arquivo *obj* de um cubo

# 3 Metodologia

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

## 3.1 Levantamento Bibliográfico

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

## 3.2 Equipamentos Utilizados

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

## 3.3 Configuração do Ambiente

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

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

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

Como pode ser visto na Figura 6, a ferramenta possui um módulo de análise dos *vertex* e *fragment shaders*, sendo possível editá-los e analisar os resultados da compilação em tempo real, além dela também gerar estatísticas.

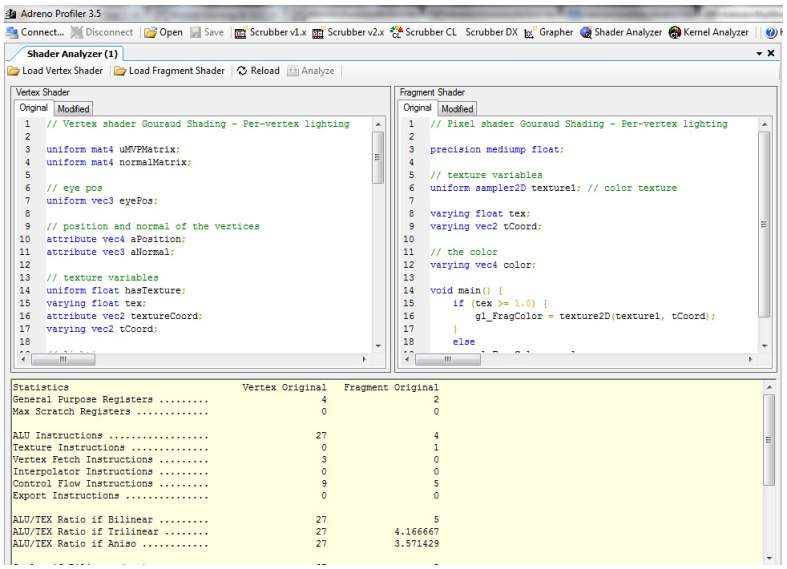


Figura 6 – Ferramenta *Adreno Profiler*: analisador de *shaders*

O módulo gráfico permite analisar algumas métricas, relacionadas ao *vertex* e *fragment shaders*, conforme ilustado na Figura 7, onde um gráfico é plotado em tempo de execução. Além disso, ela também exporta os resultados no formato CSV (*Comma-Separated Values*), que consiste em um arquivo de texto que armazena valores tabelados separados por um delimitador (vírgula ou quebra de linha). O último módulo é o chamado *Scrubber*, que provê informações detalhadas quanto ao rastreamento de uma chamada de função.

Para a automatização da plotagem dos gráficos e aplicação do método dos mínimos quadrados (descrito na Seção 2.3.2) utilizou-se a linguagem de programação Python[1] versão 2.7.3, juntamente com os pacotes *matplotlib*[2] e *numpy*[3].

O controle de versionamento do código foi feito por meio do sistema de controle de versão Git [4] utilizando o *forge* GitHub [5]. Foram criados repositórios públicos tanto para

1 http://www.python.org.br/
2 http://www.matplotlib.org/
3 http://www.numpy.org/
4 http://git-scm.com/
5 https://github.com/


Figura 7 – Ferramenta *Adreno Profiler*: visualização de métrica quadros por segundo

a implementação em *Android* [6] quanto para a automatização da análise da complexidade algorítmica [7].

## 3.4 Implementação

A fim de tornar possível a realização da análise da complexidade algorítmica experimentalmente, primeiramente focou-se na implementação dos *shaders* para plataforma *Android* utilizando a biblioteca *OpenGL ES*. Foi utilizado o paradigma de orientação a objetos, em que o diagrama de classes (Figura 8) mostra como o código foi estruturado. Este diagrama mostra um conjunto de classes e seus relacionamentos, sendo o diagrama central da modelagem orientada a objetos.

### 3.4.1 Tela de *Front-end*

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

A *Splash Activity* (Figura 10) é responsável pela visualização da tela de *loading* enquanto carrega os recursos necessários para o programa (como a leitura dos modelos tridimensionais em formato *obj* e das imagens usadas para texturização) por meio do uso de *thread*. Estes recursos são gerenciados pela classe *Resources*, que utiliza o padrão de

[6] https://github.com/campeloal/monografia_opengles
[7] https://github.com/campeloal/monografia_algorithmComplexity/

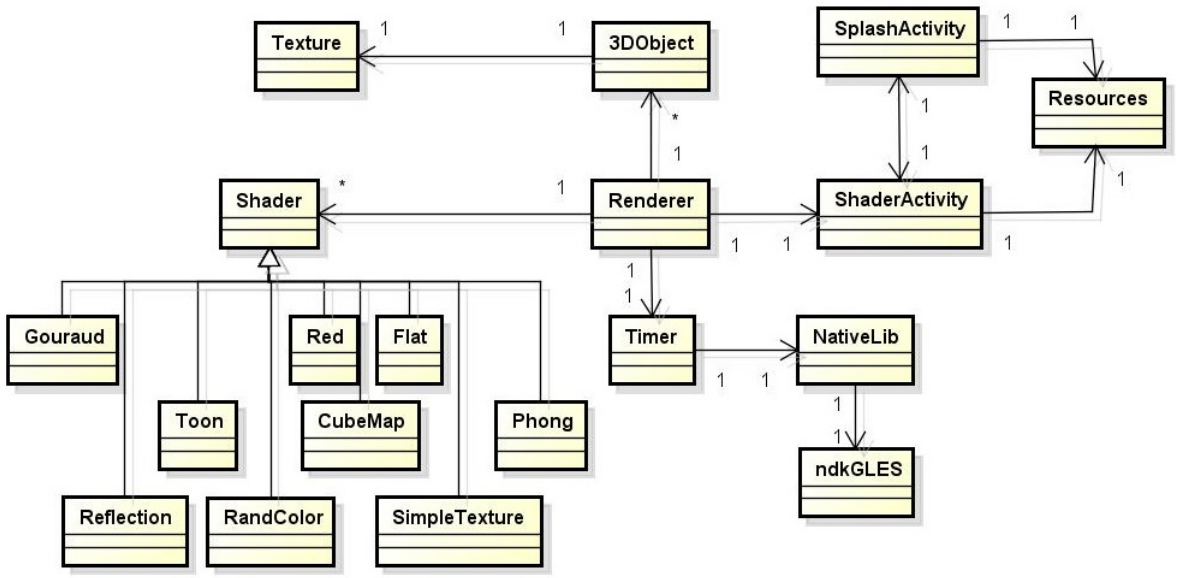


Figura 8 – Diagrama de Classe da Implementação em *Android*

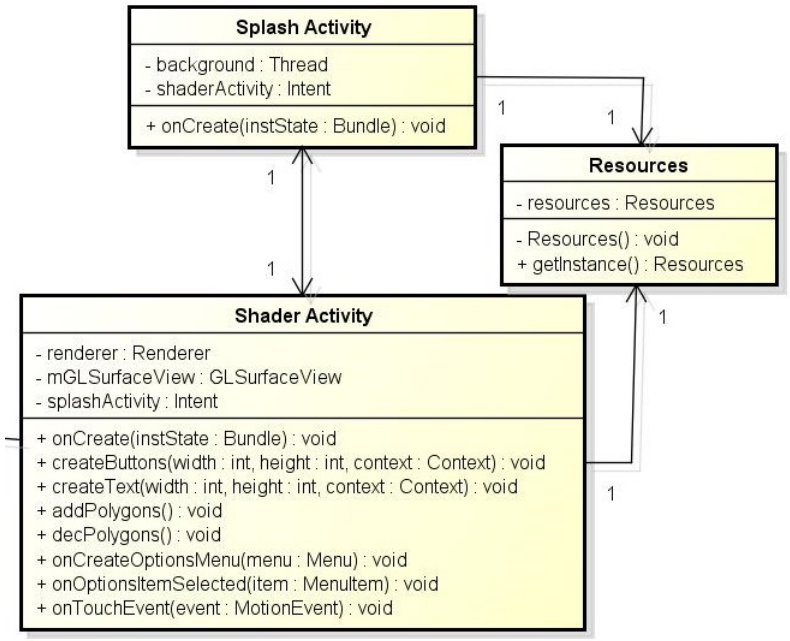


Figura 9 – Detalhamento das classes *Shader Activity*, *Splash Activity* e *Resources*

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

A *Shader Activity* (Figura 11) é responsável pela instanciação da classe *Renderer*, que renderiza os gráficos tridimensionais utilizando a biblioteca *OpenGL ES*. Além disso, ela controla os eventos de *touch*, que permitem escalar e mover o objeto, além de disponibilizar os menus que trocam de *shaders*, os botões que aumentam ou diminuem o número de polígonos, a informação do tempo de renderização e a de quantidade de polígonos. O

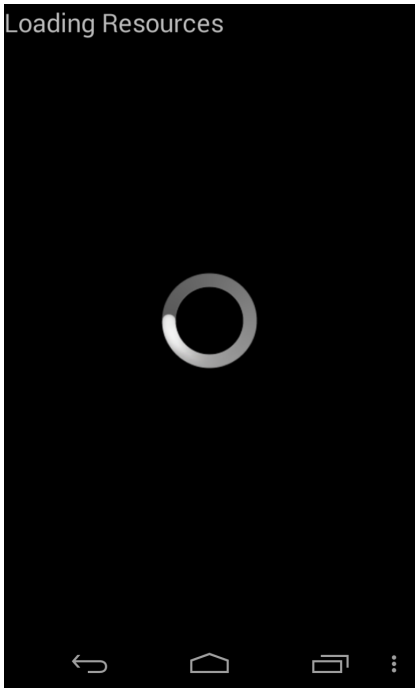


Figura 10 – Tela da *Splash Activity*

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


Figura 11 – Tela da *Shader Activity*

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

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

### 3.4.2 Objeto Tridimensional

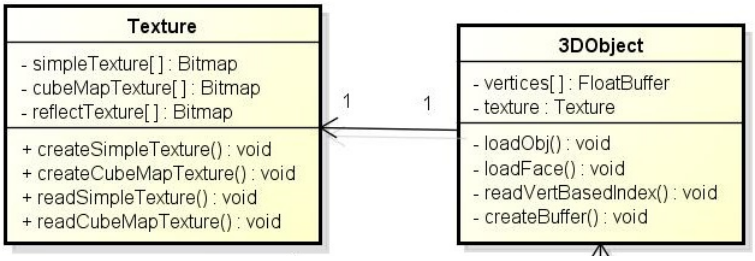


Figura 12 – Detalhamento das classes *3DObject* e *Texture*

O objeto tridimensional foi representado pela composição das classes *3DObject* e *Texture*. A classe *3DObject*, mostrada na Figura 12, é responsável por ler e interpretar os arquivos *obj* (descrito na Seção 2.4). Estes arquivos foram criados e exportados utilizando a ferramenta de modelagem tridimensional *open source* chamada Blender[8], a qual está ilustrada na Figura 13. Ela também permite modificar o número de polígonos de um modelo tridimensional por meio de modificadores chamados *Decimate* e *Subdivision Surface* (usados para diminuir e aumentar a contagem poligonal, respectivamente). Assim, foi possível variar a contagem poligonal a partir de diferentes arquivos *obj*.

Após a leitura e interpretação do arquivo *obj* foi gerado um *buffer* para armazenar os vértices de posição, normal e textura na ordem em que eles serão renderizados. Neste *buffer* cada coordenada relacionada a um vértice (posição, normal e textura) é armazenada alternadamente, como mostra a Figura 14.

A classe *Texture* gera as texturas utilizadas pelos *shaders SimpleTexture*, *CubeMap* e *Reflection* baseando-se em imagens. As imagens são criadas para cada modelo tridimensional, utilizando a técnica de *UV Mapping*, na qual mapeiam-se as coordenadas de textura para uma imagem (Figura 15). Como a orientação do eixo de coordenas $y$ da ferramenta *Blender* é diferente da *OpenG ES*, é necessário refletir a imagem neste eixo para corrigir o mapeamento.

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

[8] http://www.blender.org/

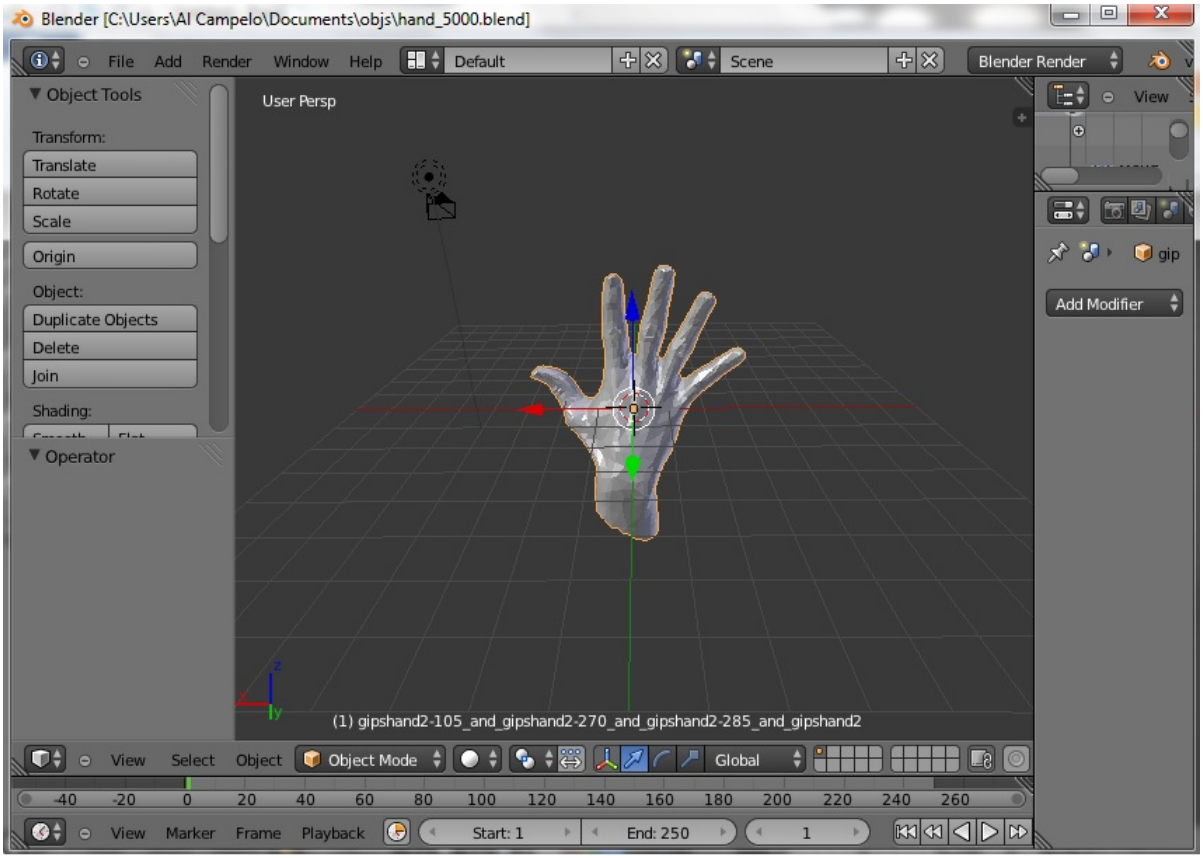


Figura 13 – Ferramenta de modelagem tridimensional

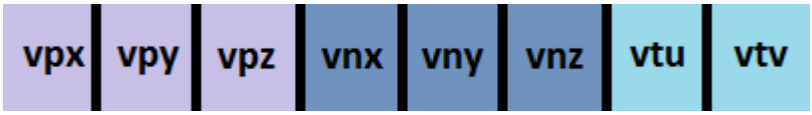


Figura 14 – Ordem das coordenadas de posição, normal e textura para um vértice

Na Figura 16 é mostrada a imagem resultante da técnica de mapeamento realizada.

### 3.4.3 Cálculo do Tempo de Renderização

A Figura 17 detalha a classe *Timer*, que realiza a média de dez medições do tempo de renderização (em nanosegundos) para cada objeto tridimensional, utilizando um *shader* específico. Cada medição é feita utilizando a linguagem C e a extensão de *OpenGL ES* chamada `GL_EXT_disjoint_timer_query` citada na Seção 3.5.1. A integração entre o código em linguagem C e o código em Java é feita por meio da classe *NativeLib*.

### 3.4.4 Renderização

A Figura 18 mostra a classe *Renderer*, responsável pela renderização, que funciona como uma controladora, sendo o ponto principal das chamadas provenientes da *view* (*ShaderActivity*) para a camada mais abaixo (*3DObject*, *Shader* e *Timer*). Ela que

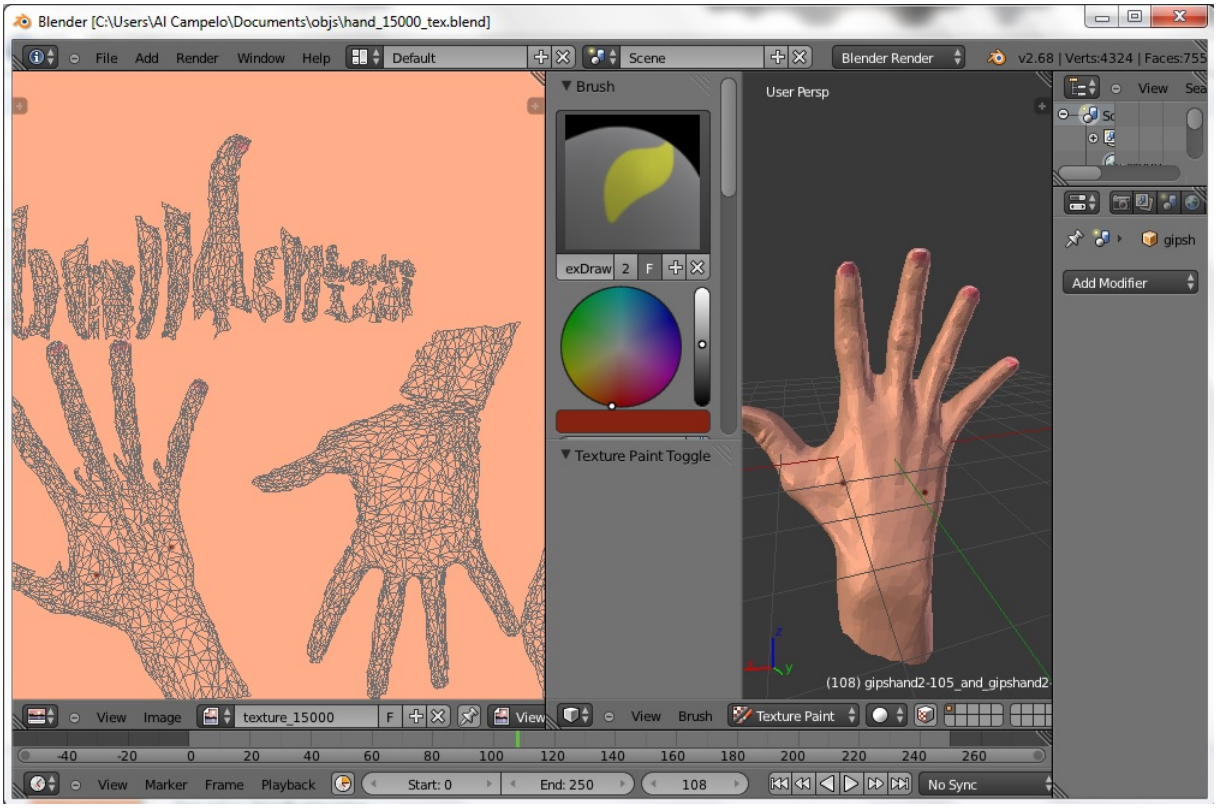


Figura 15 – Técnica de mapeamento de textura utilizada para cada modelo 3D

Figura 16 – Textura gerada a partir da técnica de mapeamento

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

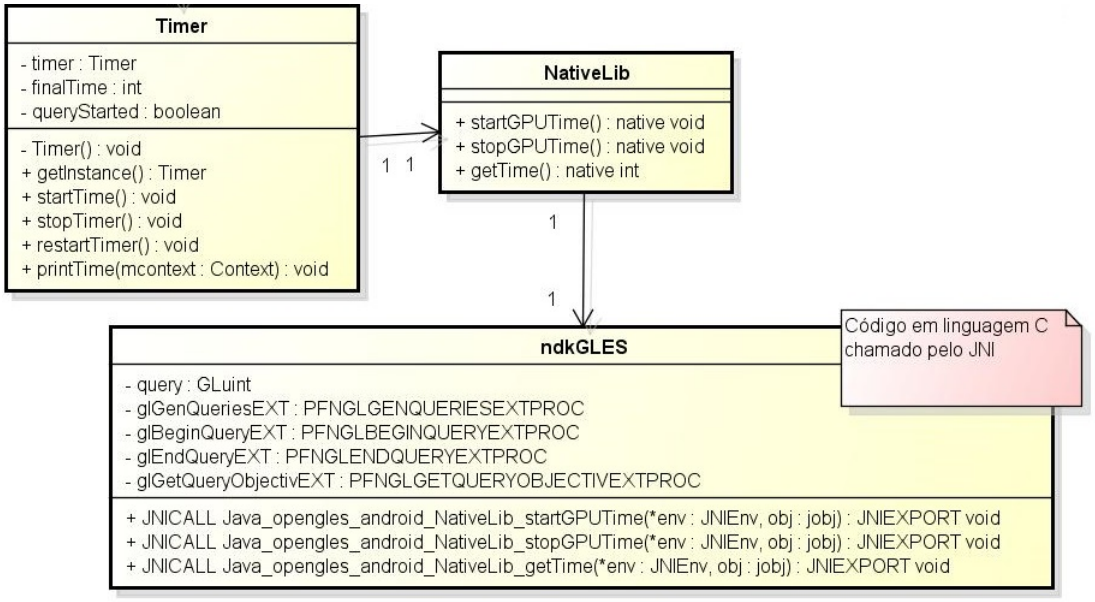


Figura 17 – Detalhamento das classes *Timer* e *NativeLib*

### 3.4.5 *Shaders*

A classe *Shader* (Figura 19) 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(Hastable params)`. Assim, todos os *shaders* que herdam desta classe, são obrigados a implementar estes métodos. O primeiro método faz o armazenamento da localização de cada variável especificada dentro do *shader*, já o segundo método inicializa estas variáveis por meio de um *hash* que é passado como um parâmetro pela classe *Renderer*. Assim, todos os *shaders* implementados herdam da classe *Shader* e implementam seus métodos abstratos. Estes *shaders* podem ser vistos na Figura 20.

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

#### 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 das normais das primitivas e então computam-se os cálculos de luz para cada fragmento, utilizando as normais interpoladas. O Código 3.1 e o Código 3.2 mostram as definições do *vertex* e *fragment shaders*, respectivamente, em que para isso é necessário definir as propriedades do material pelo programa.

Código 3.1 – *Phong Shader*: *vertex shader*

```
1 uniform mat4 uMVPMatrix;
2 uniform mat4 normalMatrix;
3 uniform vec3 eyePos;
```

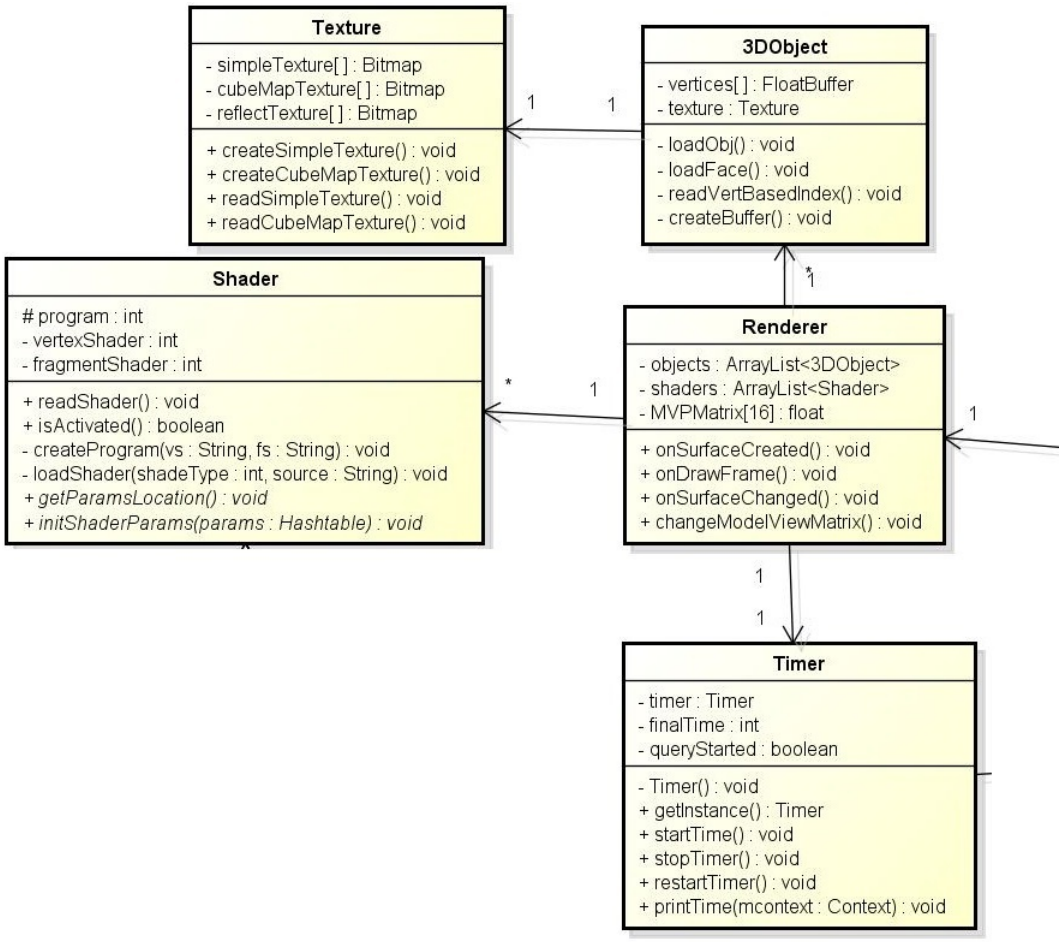


Figura 18 – Detalhamento da classe *Renderer*

```
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;
```

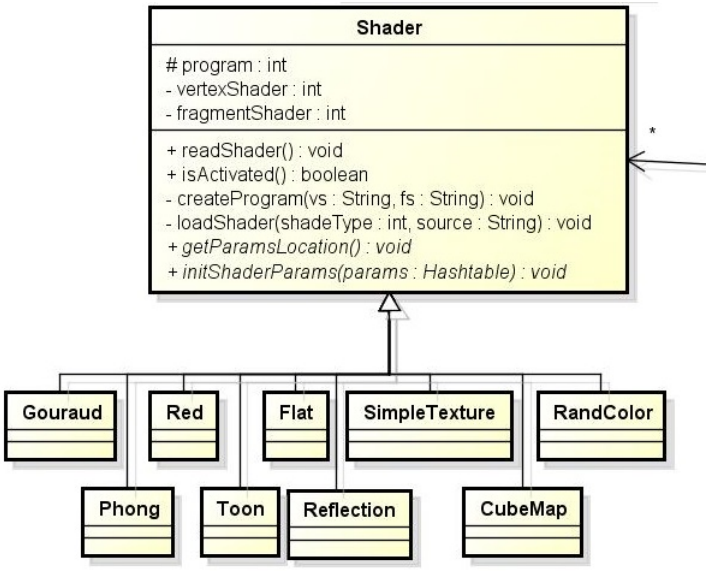


Figura 19 – Detalhamento da classe *Shader*

```
22
23          gl_Position = uMVPMatrix * aPosition;
24  }
```

Código 3.2 – *Phong Shader*: *fragment shader*

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

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

#### 3.4.5.2 *Red Shader*

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

Código 3.3 – *Red Shader*: *vertex shader*

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

Já o seu *fragment shader* (Código 3.4) estabele que todo fragmento possui a cor vermelha, por meio da palavra restrita `gl_FragColor`.

Código 3.4 – *Red Shader*: *fragment shader*

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

#### 3.4.5.3 *Toon Shader*

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

Código 3.5 – *Toon Shader*: *vertex shader*

```
1  uniform vec3 lightDir;
2  uniform mat4 uMVPMatrix;
3  attribute vec3 aNormal;
4  attribute vec4 aPosition;
5  varying float intensity;
6
```

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

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

Código 3.6 – *Toon Shader*: *fragment shader*

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

#### 3.4.5.4 *Simple Texture Shader*

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

Código 3.7 – *Simple Texture Shader*: *vertex shader*

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

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

Código 3.8 – *Simple Texture Shader*: *fragment shader*

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

#### 3.4.5.5 *CubeMap Shader*

O *CubeMap Shader* implementa a técnica também descrita na Seção 2.2. O seu *vertex shader* é simples e só define a posição do vértice (Código 3.9).

Código 3.9 – *CubeMap Shader*: *vertex shader*

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

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

Código 3.10 – *CubeMap Shader*: *fragment shader*

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

#### 3.4.5.6 *Reflection Shader*

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

Código 3.11 – *Reflection Shader*: *vertex shader*

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

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

Código 3.12 – *Reflection Shader*: *fragment shader*

```
1  varying vec3 EyeDir;
2  varying vec3 Normal;
3
4  uniform samplerCube s_texture;
5
6
7  void main()
8  {
9
```

```
10    reflectedDirection.y = -reflectedDirection.y;
11    gl_FragColor = textureCube( s_texture, reflectedDirection);
12 }
```

## 3.5 Análise da Complexidade Algorítmica Experimentalmente

A análise da complexidade algorítmica, experimentalmente, foi feita coletando-se diversas medições para cada modelo tridimensional (com diferentes quantidades de polígonos), e em seguida, plotaram-se os gráficos e eles foram ajustados à uma curva que tivesse melhor aproximação.

### 3.5.1 Medição do Tempo de Renderização Realizada pela GPU

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

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

Feita a configuração e implementação da coleta de tempo para execução da função `glDrawArrays` (responsável pela renderização), foram plotados os gráficos - para cada *shader* - do tempo em nanosegundos *versus* a quantidade de polígonos. Porém, após feita as plotagens para todos os *shaders*, foi possível perceber que as formas de todas as funções se assemelhavam a uma exponencial (como é visto na Figura 21 referente ao *phong shader*). É possível ver esta semelhança trocando-se a escala do grafico natural para logarítmica (apenas no eixo $y$), pois uma exponencial nesta escala vira uma reta (como demonstrado na Seção 2.3.2 e na Figura 21). Então, chegou-se à conclusão que ao se coletar o tempo de realização da função *glDrawArrays*, estava-se contabilizando todo o processo de renderização e não somente o de aplicação do *vertex* e *fragment shaders*.

[9] http://www.khronos.org/registry/gles/

Então, viu-se necessário a coleta de métricas específicas relacionadas ao *vertex* e *fragment shaders*.

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

Para a coleta de medições relacionadas ao *vertex* e *fragment shaders*, utilizou-se a ferramenta *Adreno Profiler*. As métricas escolhidas foram a de número de instruções por segundo por vértice e número de instruções por segundo por fragmento e elas foram coletadas para cada número específico de polígonos, sendo exportadas no formato CSV. A fim de ajustar as curvas a uma função, e consequentemente analisar a complexidade algorítmica, utilizou-se o método dos mínimos quadrados (Seção Seção 2.3.2) para uma função linear, quadrática, cúbica e exponencial, calculando-se os respectivos erros, e podendo assim, descobrir qual curva se aproxima mais.

Para automatizar estas etapas descritas anteriormente, foi feito um programa em Python que lê os arquivos CSV, faz a média das medições, plotagens dos gráficos para o *vertex* e *fragment shaders* e realiza os ajustes das curvas, calculando os erros associados e qual o menor entre eles. Ele é executado por linha de comando, em que passa-se como parâmetro o *shader* desejado (Figura 22).

O programa foi estruturado de acordo com a Figura 23, em que a classe *ReadCSV* é responsável por ler os arquivos CSV e fazer a média das métricas tanto para o *vertex shader* como para o *fragment shader*. Já a classe *PlotChart*, faz a plotagem do gráfico do número de instruções por segundo por vértice *versus* o número de polígonos e do gráfico do número de instruções por segundo por fragmento *versus* o número de polígonos. Além disso, ele também plota o gráfico original juntamente com o gráfico após a 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 qual o menor.

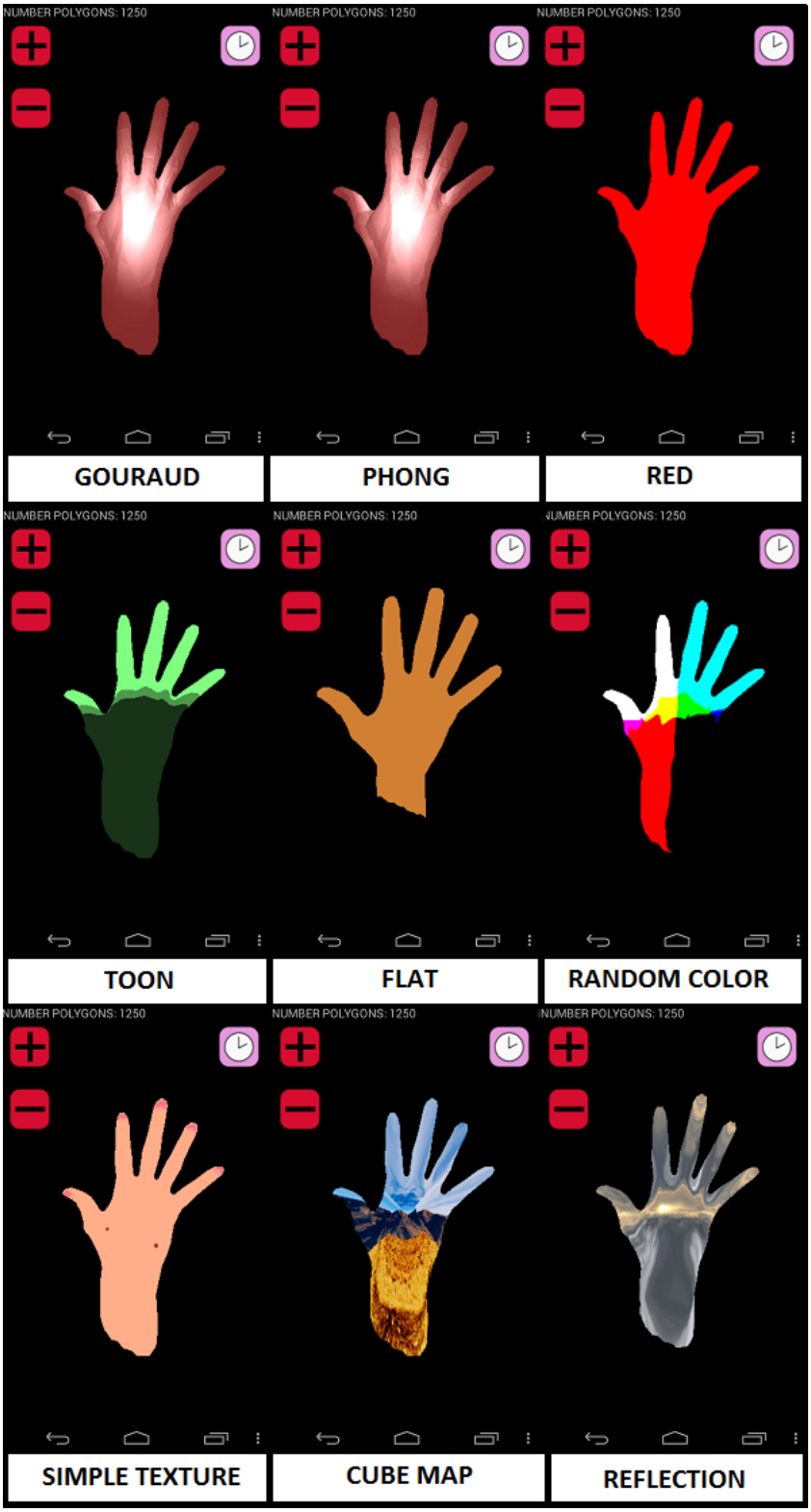


Figura 20 – *Shaders* Implementados

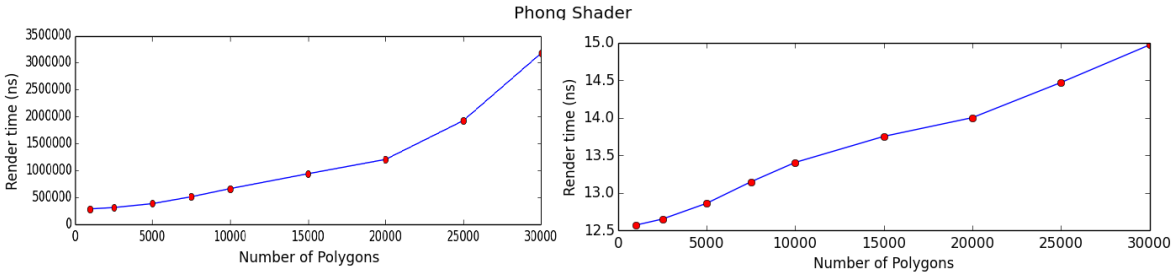


Figura 21 – Gráfico em escala natural (à esquerda) e logarítmica (à direita)

```
C:\Users\Al Campelo\Documents\UnB\Monografia\minquad>python shaderComplexity.py
reflection_
```

Figura 22 – Linha de comando para execução do programa

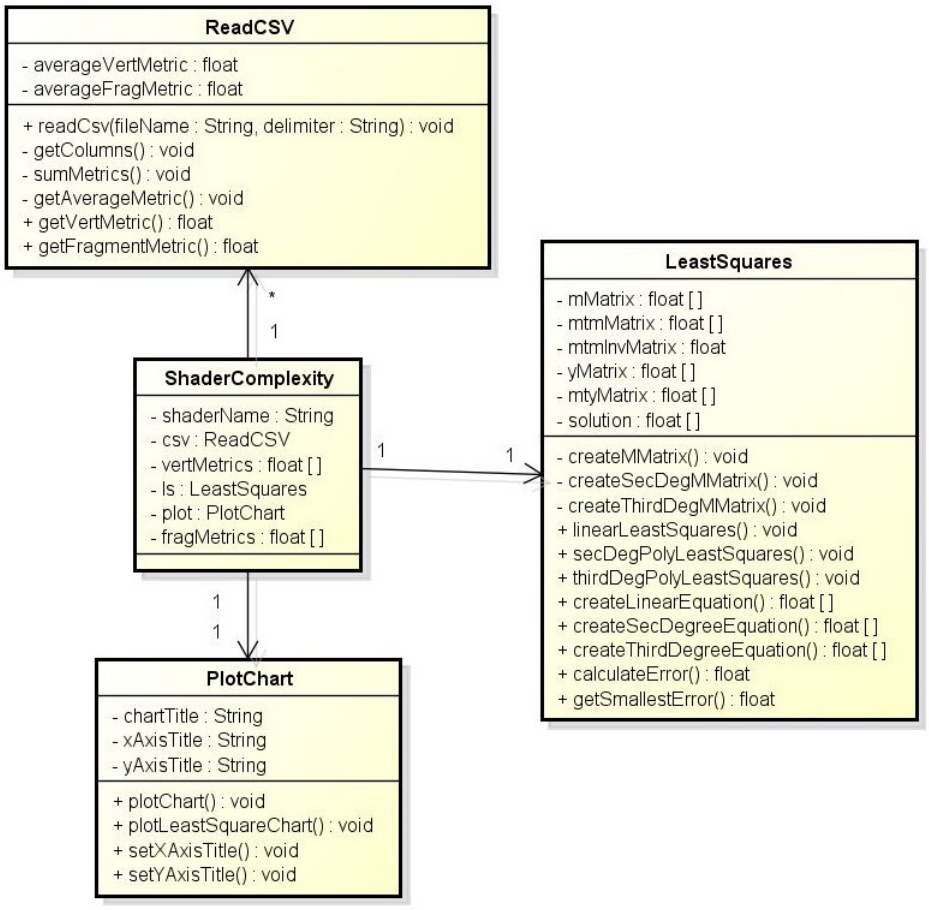


Figura 23 – Diagrama de Classes do código de automatização

# 4 Resultados

Para cada *shader* foram plotados os gráficos para as métricas relacionadas ao vértice e ao fragmento. Após as plotagens, percebeu-se que todos os gráficos de todos os *shaders* relacionados ao vértice deram uma função linear (diferindo na inclinação), e os relacionados ao fragmento deram uma curva de formato semelhante. A Figura 25 e a Figura 26 mostram os gráficos plotados, com relação ao *vertex* e *fragment shaders* de cada *shader* implementado, que demonstram a semelhança destas curvas

Os ajustes de cada curva (linear, exponencial, segundo e terceiro graus) também foram calculados e plotados (Figura 27 referente ao *Reflection Shader*), em que também avisa-se qual foi o menor erro associado, a fim de descobrir qual a curva do *fragment shader*. Pela a análise do menor erro, calculado de acordo com a Seção 2.3.2, todos os *shaders* indicaram uma curva de segundo grau.

As equações calculadas para cada *shader* (relacionadas ao vértice e fragmento) são mostradas na Tabela 7. Assim, o processo utilizado neste trabalho, da análise da complexidade algorítimica calculada de forma empírica, pode ser resumido na Figura 24. A etapa de Implementar *Shaders* pode ser feita por meio da utilização da base do projeto implementado, extendendo-se da classe *Shader* e implementando os métodos abstratos, como explicado na Seção 3.4. A etapa Realização das Medições é feita de forma manual, dependendo do *profiler* de GPU adequado para o *device* utilizado. E a etapa Plotar Gráficos, Ajustar Curvas e Obter Equações é feita através do *script* criado para o ajuste das curvas.

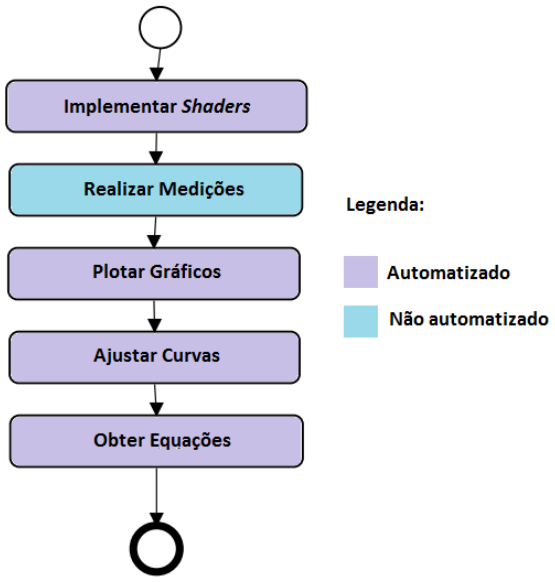


Figura 24 – Processo da Análise de Complexidade Algorítmica.

| **Nome** | ***Equação Vertex Shader*** | ***Equação Fragment Shader*** |
|---|---|---|
| *Gouraud* | $y = 19.92x10^8 + 509.43t$ | $y = 67.99x10^5 + 6076.30t - 0.014t^2$ |
| *Phong* | $y = 19.45x10^8 + 74.85t$ | $y = 20.20x10^6 + 9605.32t - 0.035t^2$ |
| *Red* | $y = 19.39x10^8 + 26.84t$ | $y = 29.95x10^5 + 5130.50t - 0.0097t^2$ |
| *Toon* | $y = 19.45x10^8 + 100.35t$ | $y = 38.25x10^5 + 5347.86t - 0.011t^2$ |
| *Flat* | $y = 19.39x10^8 + 26.77t$ | $y = 25.02x10^5 + 4285.80t - 0.0091t^2$ |
| *Random Color* | $y = 19.45x10^8 + 74.04t$ | $y = 84.32x10^5 + 6930.50t - 0.021t^2$ |
| *Simple Texture* | $y = 19.42x10^8 + 49.97t$ | $y = 31.76x10^5 + 5137.26t - 0.0099t^2$ |
| *CubeMap* | $y = 19.44x10^8 + 83.38t$ | $y = 33.14x10^5 + 5109.61t - 0.0094t^2$ |
| *Reflection* | $y = 19.62x10^8 + 289.76t$ | $y = 83.92x10^5 + 6494.94t - 0.017t^2$ |

Tabela 7 – Equações relacionadas ao *vertex shader* e *fragment shader*

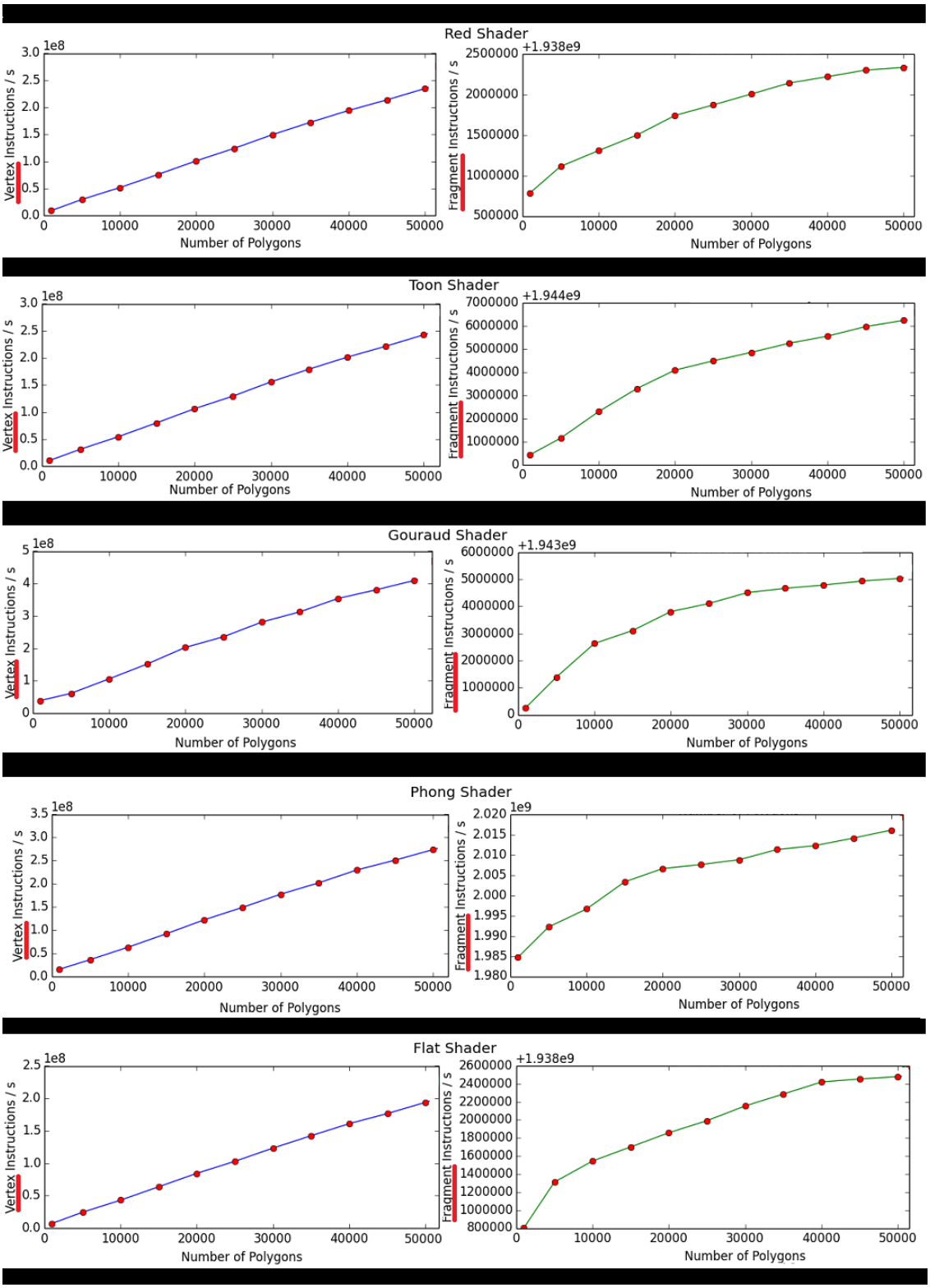


Figura 25 – Gráficos: *Red Shader*, *Toon shader*, *Gouraud Shader*, *Phong Shader* e *Flat Shader*

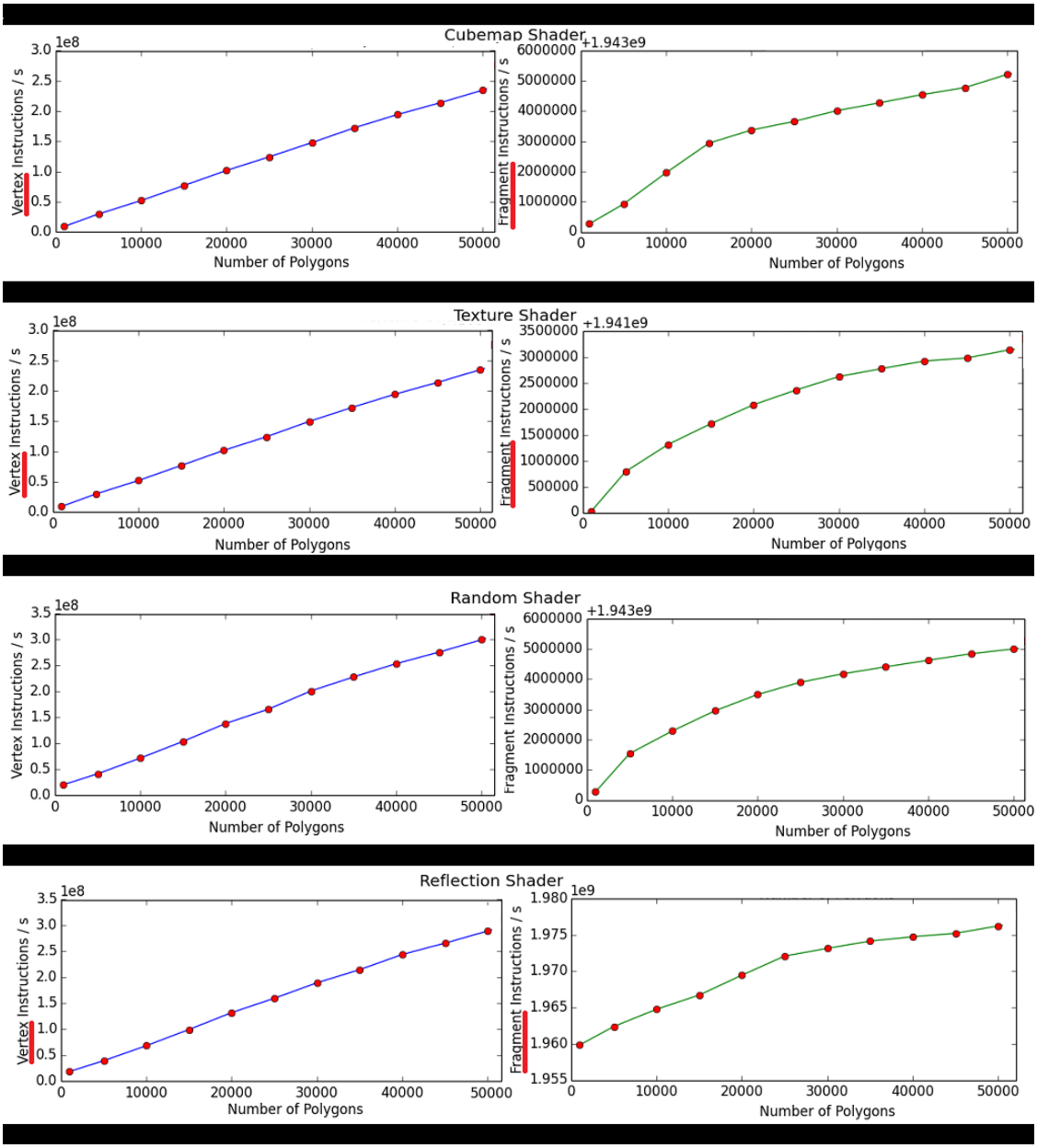


Figura 26 – Gráficos: *Cubemap Shader*, *Texture Shader*, *Random Shader*, *Reflection Shader*

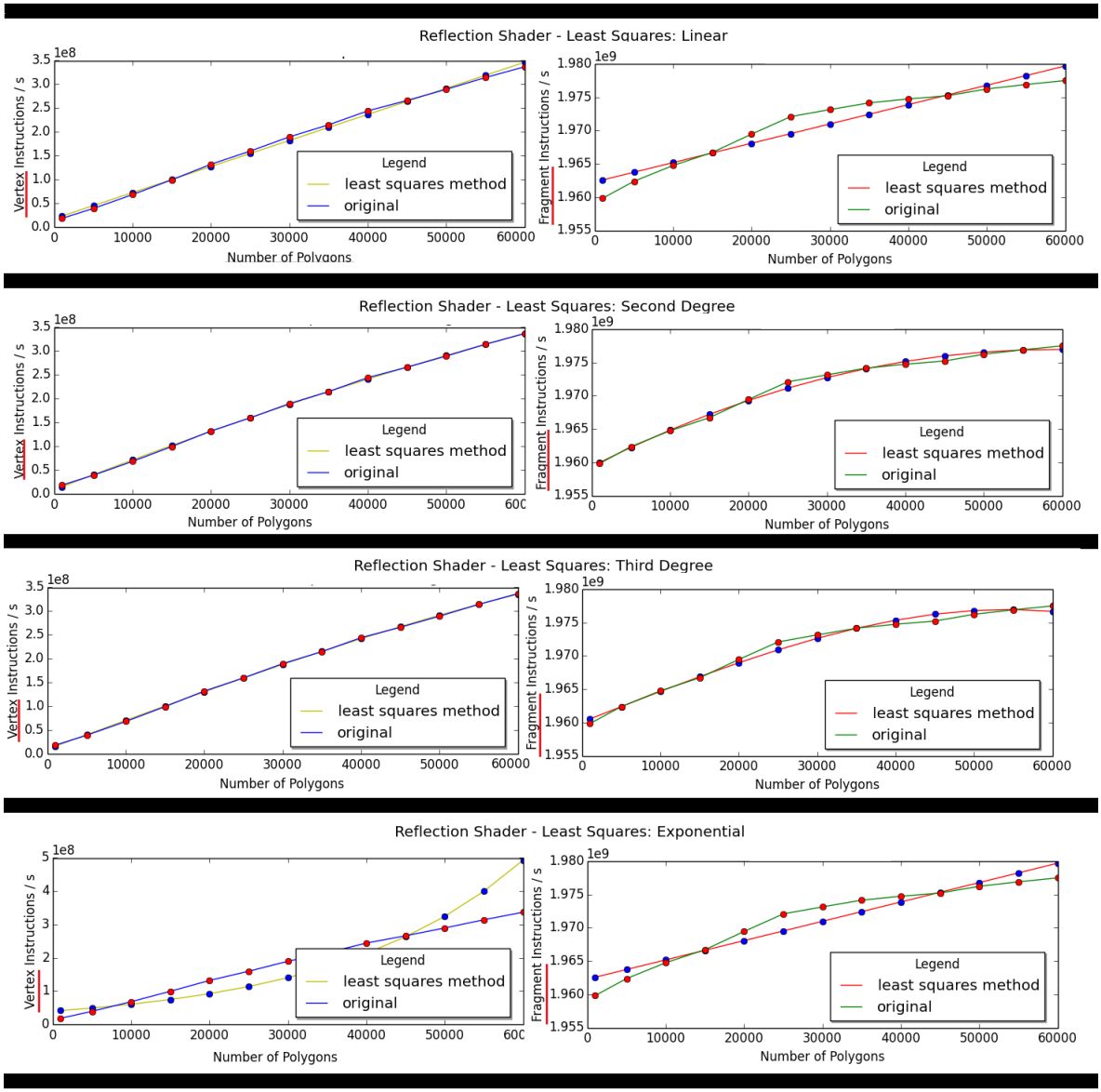


Figura 27 – Ajustes linear, para função de segundo, terceiro graus e exponencial

# 5 Conclusão

Por meio do trabalho realizado, foi possível concluir que o processo de renderização como um todo, calculado por meio do tempo de renderização da GPU, terá sempre como complexidade algorítmica uma exponencial para qualquer *shader* (variando somente os coeficientes da função). Já o processo relacionado ao *vertex shader*, por meio dos experimentos realizados, foi possível perceber que a complexidade algorítmica sempre será linear e o relacionado ao *fragment shader* sempre será um polinômio de segundo grau.

Analisando a teoria do processo de renderização da *OpenGL* este resultado se confirma, pois o programa do *vertex shader* é utilizado para cada vértice (sendo de ordem linear). O do *fragment shader*, por sua vez, é de ordem do segundo grau, já que seu programa é usado para cada fragmento (sendo uma matriz). Além disso, ao analisar a documentação da GLSL[1], percebe-se que um fragmento (ou vértice) específico não tem conhecimento dos seus fragmentos (ou vértices) vizinhos, então não é possível que a complexidade cresça de um polinômio de segundo grau para um de terceiro, por exemplo. Porém este resultado não é tão óbvio, pois ao analisar o programa do *vertex shader* do Código 5.1, por exemplo, há somente uma atribuição, induzindo o programador a achar que a complexidade é constante. E por meio deste trabalho foi possível perceber que não é.

Código 5.1 – Exemplo de programa do *vertex shader*

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

Assim, como todos os *shaders* (do mesmo tipo) possuem a mesma complexidade algorítmica, uma forma de comparar o desempenho entre eles 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

[1] http://www.opengl.org/documentation/glsl/

). Isto pode ser feito para comparar diferentes *shaders* ou para saber o quanto um *shader* foi otimizado (comparando-se o anterior e o atual).

Outra contribuição importante foi quanto à automatização da maior parte deste processo de análise da complexidade algorítimica, como a estrutura para aplicação dos *shaders*, média das medições, plotagem, ajuste das curvas e cálculo das funções. Assim, torna a sua execução mais rápida e confiável.

# Referências

ARNAU, J.; PARCERISA, J.; XEKALAKIS, P. Teapot: A toolset for evaluating performance, power and image quality on mobile graphics systems. 27th Int. Conf. on Supercomputing, p. 37–46, 2013. Citado na página 17.

BROTHALER, K. *OpenGL ES 2 for Android*: A quick-start guide. 1. ed. Dallas, Texas: The Pragmatic Bookshelf, 2013. Citado na página 63.

DROZDEK, A. *Estrutura de Dados e Algoritmos em C++*. 2. ed. São Paulo, São Paulo: Cengage Learning, 2002. Citado na página 26.

GUHA, S. *Computer Graphics Through OpenGL*: From theory to experience. 1. ed. Boca Raton, Florida: CRC Press, 2011. Citado 3 vezes nas páginas 24, 25 e 26.

JACKSON, W. *Learn Android App Development*. 1. ed. New York, New York: Apress, 2013. Citado 2 vezes nas páginas 24 e 33.

LEITHOLD, L. *O Cálculo com Geometria Analítica*. 3. ed. São Paulo, São Paulo: Harbra, 1994. Citado na página 28.

MOLLER, T. A.; HAINES, E.; HOFFMAN, N. *Real-Time Rendering*. 2. ed. Boca Raton, Florida: CRC Press, 2008. Citado na página 21.

NADALUTTI, D.; CHITTARO, L.; BUTTUSSI, F. Rendering of x3d content on mobile devices with opengl es. Proc. Of 3D technologies for the World Wide Web, Seção Mobile devices, p. 19–26, 2006. Citado na página 17.

QUALCOMM, D. N. *Mobile Gamimg 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 32.

RORRES, A. *Álgebra Linear com Aplicações*. 8. ed. Porto Alegre, Rio Grande do Sul: Bookman, 2001. Citado na página 27.

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.

SEDGEWICK, R. *Algorithms in C*. 1. ed. Westford,Massachusetts: Addison-Wesley, 1990. Citado na página 27.

SHERROD, A. *Game Graphics Programming*. 1. ed. Boston, Massachusetts: Course Technology, 2011. Citado 2 vezes nas páginas 17 e 29.

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 primitvas 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 28, tem-se dois triângulos e quatro vértices definidos (dois vértices são compartilhados). Assim, pode-se definir um vetor com os vértices [v0, v1, v2, v3] e um vetor de índices [0, 3, 1, 0, 2, 3], que determina a ordem em que os vértices devem ser renderizados.

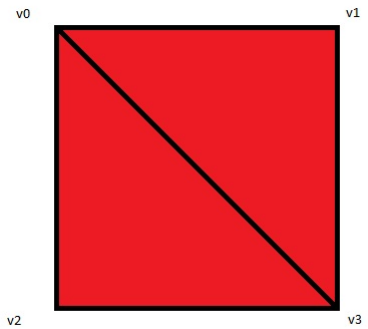


Figura 28 – Vértices do quadrado constituído de dois triângulos

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 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 29 ilustra esta ideia.

## A.4 Montagem das Primitivas

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

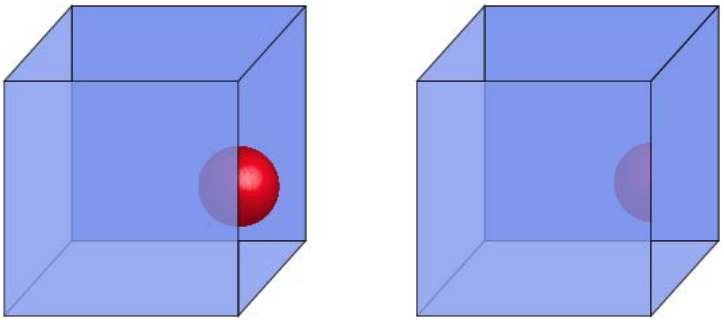

Figura 29 – Antes do recorte (cubo de visualização esquerdo) e depois do recorte (cubo de visualização direito)

## A.5 Conversão e Interpolação dos Parâmetros das Primitivas

Nesta etapa que ocorre o procedimento conhecido como rasterização. Nela, cada primitiva é transformada em fragmentos, cuja entrada é formada pelas primitivas recortadas e as coordenadas ainda tridimensionais. Assim, esta etapa tem como finalidade mapear as coordenadas tridimensionais em coordenadas de tela. Para isto, o centro de um *pixel* (*picture element*) é igual à coordenada 0,5. Então, *pixels* de [0; 9] equivalem à cobertura das coordenadas de [0,0; 10,0). E os valores dos *pixels* crescem da esquerda para a direita e de cima para baixo. Também é feita a configuração dos triângulos, em que dados são computados para as superfícies dos triângulos. Eles serão utilizados para a conversão dos dados vindos do processo de geometria (coordenadas e informações provenientes do *vertex shader*) em *pixels* na tela e também para o processo de interpolação. Assim, é checado se cada um dos *pixels* está dentro de um triângulo ou não. Para cada *pixel* que sobrepõe um triângulo, um fragmento é gerado, conforme mostrado na Figura 30. Cada fragmento tem informações sobre sua localização na tela, no triângulo e sua profundidade, e as propriedades dos fragmentos dos triângulos são geradas usando dados interpolados entre os três vértices do triângulo.

## A.6 Processamento dos Fragmentos

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

## A.7 Processamento das Amostras

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

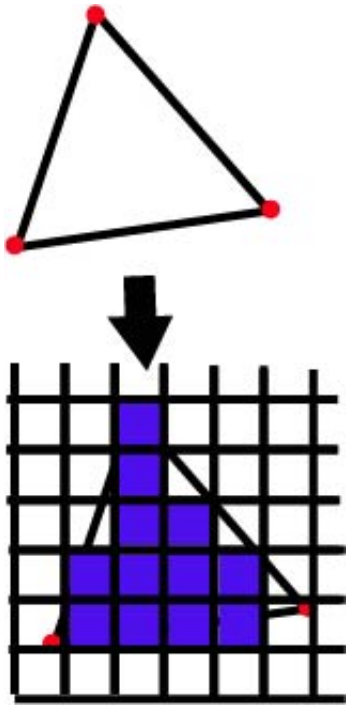

Figura 30 – Travessia de triângulos: fragmentos sendo gerados

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