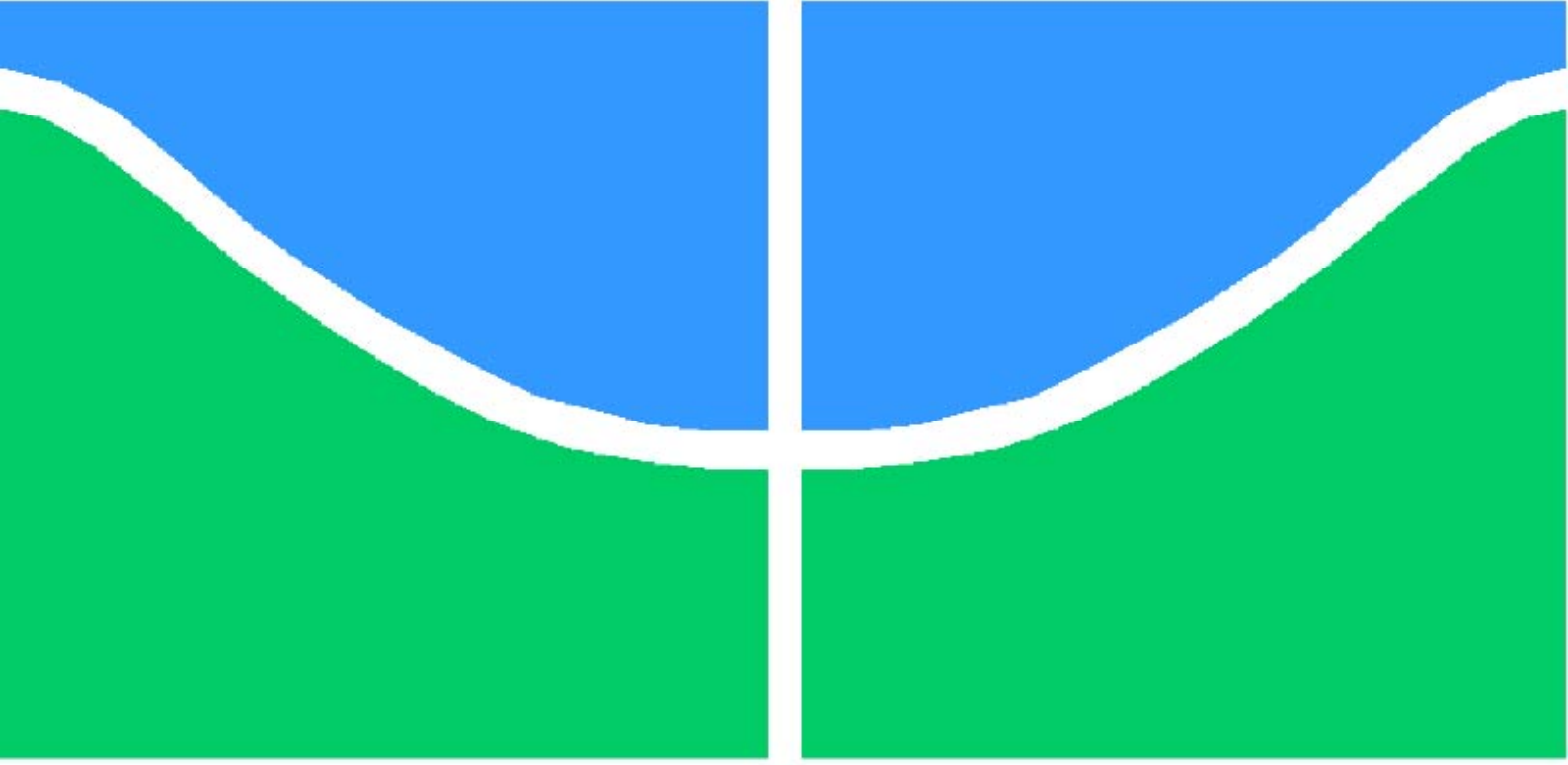

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Engenharia de *Software*

Implementação de *Shaders* para Plataforma *Android*

Autor: Aline de Souza Campelo Lima

Orientador: 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: 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*.

Trabalho aprovado. Brasília, DF, 01 de junho de 2013:

Dr. Edson Alves da Costa Júnior
Orientador

Titulação e Nome do Professor
Convidado 01
Convidado 1

Titulação e Nome do Professor
Convidado 02
Convidado 2

Brasília, DF
2013

Resumo

Palavras-chaves:

Abstract

Key-words:

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*

Sumário

1 Introdução

1.1 Contextualização

1.2 Justificativa

1.3 Delimitação do Assunto

O tema consiste no desenvolvimento de *shaders* aplicados em objetos tridimensionais - com número de polígonos variante - no qual em seguida renderiza-se a cena e são coletadas 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 posteriormente poder aplicar o método dos mínimos quadrados (como explicado na seção 2 Referencial Teórico), a fim de estimar o número de quadros por segundo de um *shader* específico dado um número n de polígonos, baseando-se na curva obtida experimentalmente pelos gráficos.

Como visto na seção 5 Referencial Teórico, 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 é factível dentro do prazo estipulado, primeiramente desenvolveu-se um *shader* no qual se utiliza a técnica *Gouraud Shading* (como explicado na seção 2 Referencial Teórico) aplicado em um objeto tridimensional, o octaedro, usando a linguagem Java (padrão do *Android*). O mesmo programa também foi feito 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 (discutido na seção 5 Resultados Alcançados).

1.4 Objetivos Gerais

1.5 Objetivos Específicos

2 Referencial Teórico

2.1 Plataforma *Android*

O *Android* começou a ser desenvolvido em 2003 na empresa de mesmo nome, fundada por Andy Rubin, na 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 hoje é o sistema operacional para *mobile* mais utilizado.

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. O *Android* também possui um mercado centralizado em cada aparelho (*tablet* ou *smartphone*) chamado *Google Play*, facilitando a publicação de aplicativos. O *Android* possui diferentes versões, sendo elas mostradas na Tab. (1) abaixo. As versões mais novas possuem mais *features* que as anteriores: a versão *Jelly Bean*, por exemplo, possui a busca por voz que a versão *Ice Cream Sandwich* não possuía.

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	*Jelly Bean*
4.4	4.4 *KitKat*

Tabela 1 – Versões da plataforma *Android*

Com o intuito de desenvolver para a plataforma *Android*, uma das alternativas é utilizar a ferramenta *Eclipse* (Fig. 1) , que é um *Integrated development environment* (IDE) *open source*. Adicionalmente, é preciso, de acordo com (JACKSON, 2013), instalar o *Android Software Development Kit* e o *plugin Android Development Tools* (ADT), que permitem desenvolver e depurar aplicações pra *Android*. Outra alternativa é utilizar o *Android Studio*, lançado recentemente (2013) pela empresa *Google*, que já vem com to-

dos os pacotes e configurações necessárias para o desenvolvimento, incluindo o *Software Development Kit* (SDK), as ferramentas e os emuladores.

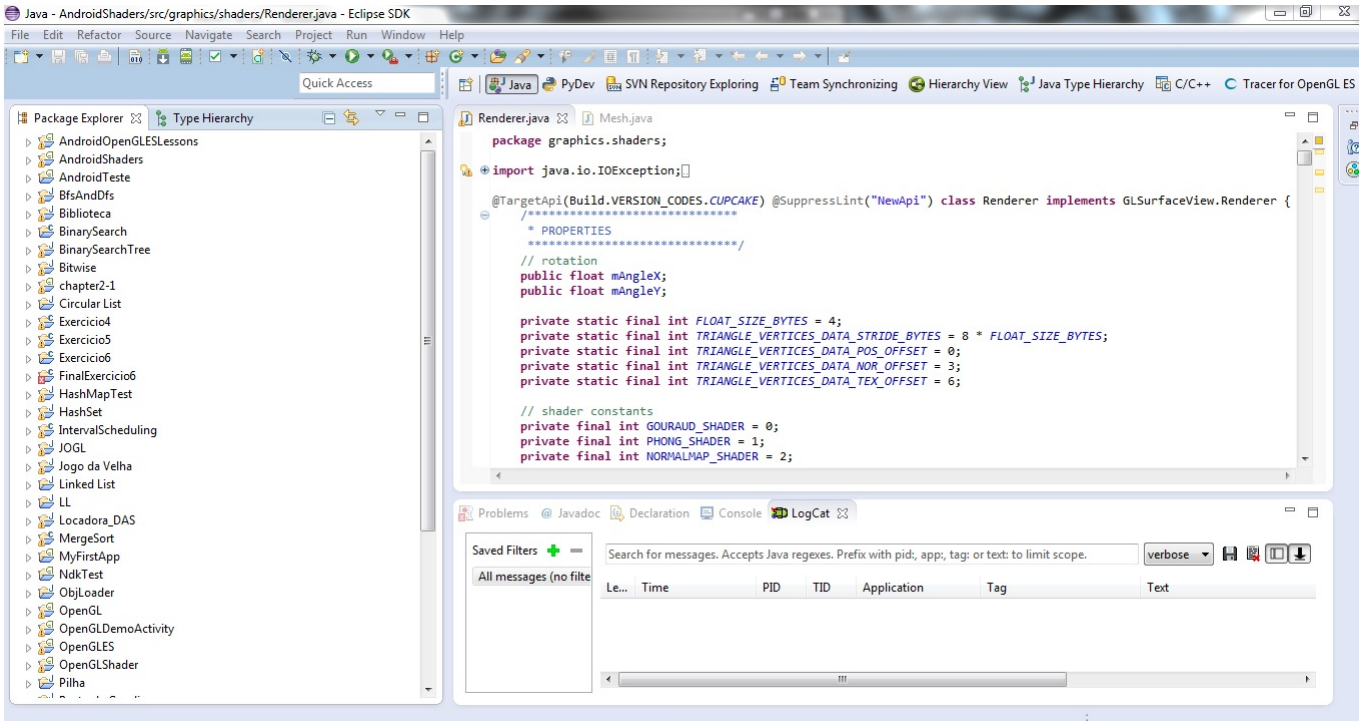

Figura 1 – Ambiente de desenvolvimento *Eclipse*

2.2 Bibliotecas Gráficas

2.2.1 *OpenGL*

A *OpenGL* é uma *Application Programming Interface* (API) utilizada em computação gráfica para modelagem tridimensional lançada em 1992 e segundo (WRIGHT et al., 2008), sua precursora foi a biblioteca *Integrated Raster Imaging System Graphics Library* (Iris GL) da empresa *Silicon Graphics*. Ela é uma API procedural, na qual é preciso descrever os passos que involvem diversos comandos *OpenGL* necessários para se chegar no efeito visual desejado.

Ela possui comandos para desenho de primitivas (como linhas e pontos, por exemplo), texturização, transparência, animação, entre outros efeitos especiais. Porém, ela não possui funções de gerenciamento de janela, eventos de *input* (de *mouse* e teclado, por exemplo) ou leitura e escrita de arquivos: o próprio programador é responsável por configurar o ambiente necessário para a *OpenGL* desenhar em uma janela (seja para *Microsoft Windows*, *Mac OS* ou *Unix*, por exemplo).

A *OpenGL* possui quatro versões, sendo a mais atual a 4.4. As principais modificações ocorreram entre a 1.x para 2.x - que permitiu o uso de *shaders* e *pipeline* de

renderização programável - e entre a 2.x e 3.x, que deprecia as funções fixas (que serão removidas nas versões posteriores).

2.2.2 *Glut*

Visando a portabilidade e abstração do sistema operacional, a biblioteca *OpenGL Utility Toolkit* (Glut) foi criada por Mark Kilgard, enquanto ele ainda trabalhava na empresa *Silicon Graphics*. Ela facilita a utilização de janelas e *input*, integrando as janelas do sistema operacional subjacente com a *OpenGL* de forma portável entre diferentes sistemas operacionais. Embora possua limitações de *Graphical User Interface* (GUI), de acordo com (WRIGHT et al., 2008), ela é simples de ser utilizada.

2.2.3 *OpenGL ES*

A *OpenGL for Embedded Systems* (*OpenGL ES*) foi lançada em 2003, e como citado em (GUHA, 2011), atualmente é uma das API's mais populares para programação de gráficos tridimensionais em pequenos *devices*, sendo adotada por diversas plataformas como *Android*, *IOS*, Nintendo DS e *Black Berry*. Segundo (ANGEL; SHREINER, 2012b), 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 e a 3.x, que é completamente compatível com a *OpenGL* 4.3.

2.3 Processo do *Rendering Pipeline*

Uma cena é composta por objetos, que por sua vez são compostos por primitvas como triângulos, quadrados, linhas, por exemplo, 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 pode ser divido em dois processos principais: o de geometria e o de rasterização. Segundo (MOLLER; HAINES; HOFFMAN, 2008), o processo de geometria pode ser divido nas etapas mostradas na Fig. 2.

Figura 2 – Etapas do processo de geometria

Na etapa Transformações de Modelagem e Visualização, as coordenadas do objeto são transformadas, de forma que ele possa ser posicionado, orientado e tenha um tamanho determinado. Após essa etapa, é dito que o objeto está localizado no espaço do mundo e é aplicada a transformação de visualização, que tem como objetivo estabelecer a câmera na origem, mirando em direção ao eixo z negativo.

A próxima etapa é a de *Vertex Shading*, responsável por modelar parte dos efeitos (a outra parte é feita durante a rasterização), pois renderizar somente a forma e posição não é suficiente. Estes efeitos incluem os materiais dos objetos, como também os efeitos da luz, podendo ser modelados de diferentes formas, como representações de descrições físicas. Muitos dados sao armazenados em cada vértice, como a sua localização e normal, por exemplo. Assim, os resultados do *vertex shading* são mandados para o estágio de rasterização para serem interpolados.

A 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 e linhas paralelas tendem a convergir no horizonte. Ela resulta em um tronco de pirâmide com base retangular.

Somente as primitivas gráficas que se encontram dentro do volume de visualização que serã renderizadas. Assim, o recorte (chamado de *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 Fig. 3 mostra esta ideia.

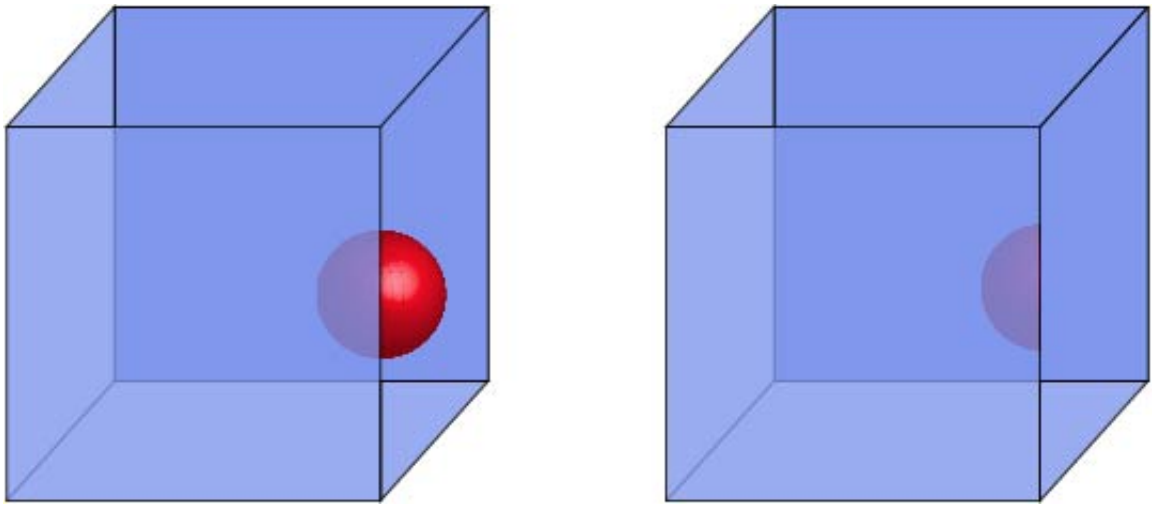

Figura 3 – Antes do recorte (cubo de visualização esquerdo) e depois do recorte (cubo de visualização direito)

A última etapa de geometria é a de mapeamento na tela, em que a entrada são as primitivas recortadas e as coordenadas ainda são tridimensionais. Assim, esta etapa tem como finalidade mapear as coordenadas tridimensionais em coordenadas de tela. Para isto, o centro de um *picture element* (*pixel*) é igual a 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.

Terminado o processo de geometria, o próximo a ser feito é o de rasterização, em que seu objetivo é computar e definir as cores para cada *pixel*. Este processo pode ser dividido em quatro etapas mostradas na Fig. 4.

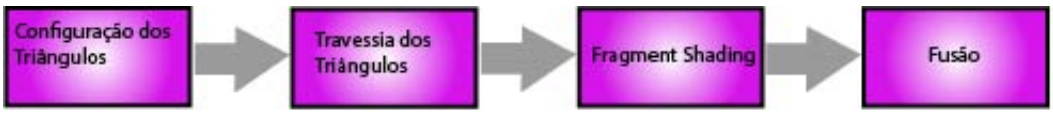

Figura 4 – Etapas do processo de rasterização

Na etapa de Configuração dos Triângulos, as diferenciais e outros dados são computados para as superfícies dos triângulos. Estes dados serão utilizados para a conversão dos dados vindos do processo de geometria (coordenadas e suas informações provenientes do *vertex shader*) em *pixels* na tela e também para o processo de interpolação.

A Travessia de Triângulos checa se cada um dos *pixels* está dentro de um triângulo ou não. Para cada *pixel* que sobrepõe um triângulo, um fragmento é gerado como mostrado na Fig. 5. Cada fragmento tem a informação 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.

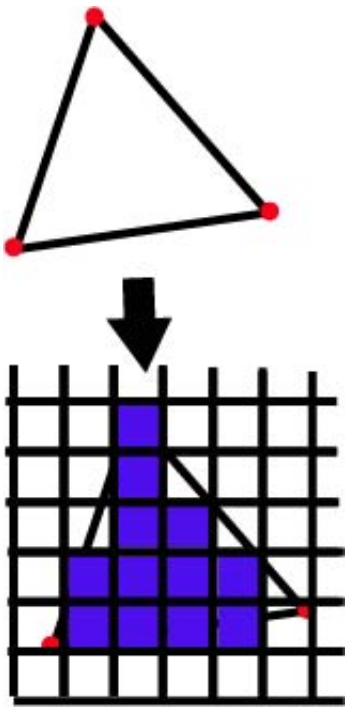

Figura 5 – Travessia de triângulos: fragmentos sendo gerados

As computações por *pixel* são calculadas durante o *fragment shading*, em que o resultado é uma ou mais cores a serem passadas para o próximo estagio. Muitas técnicas

podem ser aplicadas durante esta etapa e uma das mais importantes é a de texturização (que aplica no fragmento do objeto parte de uma imagem).

A informação relacionada com cada *pixel* é armazenada no *color buffer*, que é um *array* de cores. Assim, a última etapa é a de fusão, que é responsável por combinar a cor do fragmento gerada pelo estágio anterior com a cor armazenada no *buffer*. Ela também é responsável pela visibilidade, 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 esta 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.

2.4 Renderizando Modelos Tridimensionais

2.4.1 Função Fixa

As versões da *OpenGL* anteriores a 3.0 e a versão 1.0 da *OpenGL ES* permitem a utilização de funções fixas para a renderização, ou seja, as funções da API em *pipeline* estático. Uma dessas operações é para a de desenho de objetos, em que se utiliza a chamada *glVertex3f(float x, float y, float z)* (entre as declarações *glBegin(Glenum mode* e *glEnd()*). *Mode* diz qual primitiva deve ser utilizada, como por exemplo, linhas, pontos e triângulos. Para se definir a cor, utiliza-se o *glColor3f(float r, float g, float b)*, em que os argumentos são as coordenadas do sistema de cores Red Green and Blue (RGB). Outras funções existentes são as de definição da projeção, utilizando *glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far* para projeções ortográficas ou *glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far* para perspectiva. O significados dos parâmetros, segundo (GUHA, 2011), podem ser vistos nas Fig. 6.

2.4.2 *Vertex Array*

Um modelo tridimensional pode ser descrito por uma lista de vértices e uma lista de índices. Na Fig. 7, 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 diz a ordem que os vértices devem ser renderizados.

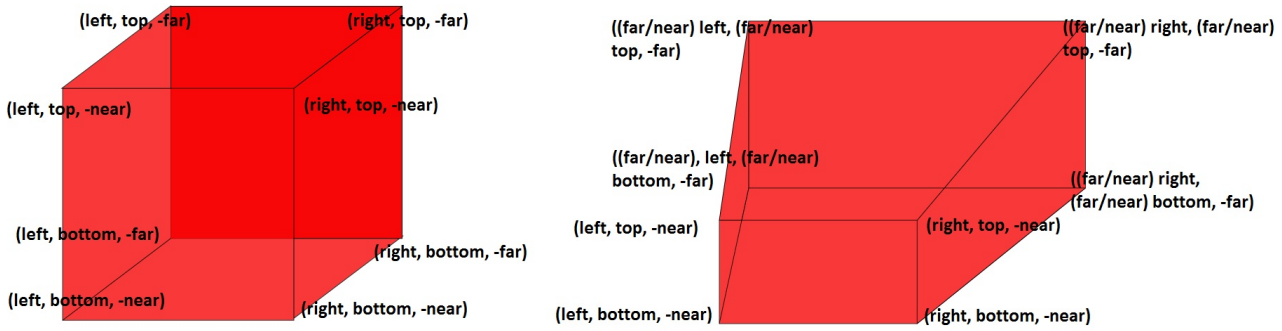

Figura 6 – Projeção Ortográfica: parâmetros

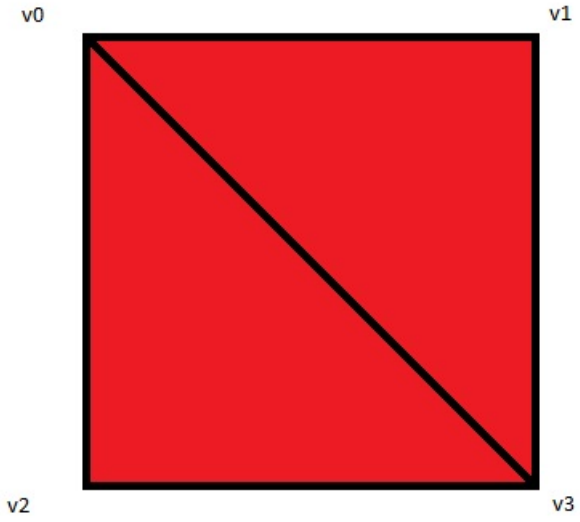

Figura 7 – Vértices do quadrado constituído de dois triângulos

Segundo (GUHA, 2011), as coordenadas dos vértices podem ser armazenadas em um vetor e passadas para a *OpenGL* como um ponteiro. O vetor de vértices é ativado com a chamada *glEnableClientState(GL VERTEX ARRAY)* e o ponteiro é definido através da função *glVertexPointer(size, type, stride, *pointer)*. O parâmetrro *pointer* é o endereço de onde começa o vetor, *type* é o tipo dos dados, *size* é o número de valores por vértice e *stride* é o *offset* em *bytes* entre o início dos valores para sucessivos vértices (zero indica que os valores para os vértices sucessivos não estão separados). Com o *stride* é possível armazenar os vértices de posição e normais em único vetor, por exemplo.

Finalmente, a renderização pode ser feita por meio da chamada *glDrawElements(primitive, count, type, *indices)*, em que *primitive* é a primitiva geométrica (pontos, linhas, triângulos, por exemplo), *type* é o tipo de dado, *indices* é o vetor de índices e *count* é o número de índices a serem usados.

Dessa forma, os dados são definidos em apenas um local, podendo ser utilizados

em vários locais do código, evitando redundância e também diversos dados (como coordenadas de posição, textura, normais, por exemplo) podem ser definidos em um único vetor, sendo mais eficiente.

2.4.3 *Vertex Object Buffer*

A ideia do *Vertex Object Buffer* é a mesma do *Vertex Array*, porém de acordo com (BROTHALER, 2013), o *driver* gráfico pode optar por colocá-lo diretamente na memória da GPU, melhorando o desempenho para objetos que não são modificados com muita frequência.

Primeiramente é necessário criar um novo *buffer* e para isso, de acordo com (ANGEL; SHREINER, 2012a), é necessário utilizar a chamada*glGenBuffers()*. Feito isto, vincula-se o vetor ao *buffer* com a fução *glBindBuffer(GLenum target, GLint id)*, em que *target* é o tipo do *buffer* (GL ARRAY BUFFER, por exemplo) e *id* é o identificador. Para fazer uma cópia dos dados do vetor ao *buffer*, utiliza-se a *glBufferData(GLenum target, GLsizeptr size, const GLvoid *data, GLenum usage)*, em que *target* é o tipo, *size* é o tamanho em *bytes* do *buffer*, *data* é o ponteiro para os dados que serão copiados e *usage* é o padrão de utilização dos dados. Os padrões e seus significados podem ser vistos na Tab. (2.

Padrão	Significado
GL STREAM DRAW	O objeto será modificado apenas uma vez e usado poucas vezes
GL STATIC DRAW	O objeto será modificado uma vez, mas será usado várias vezes
GL DYNAMIC DRAW	O objeto será modificado e usado várias vezes

Tabela 2 – Palavras-chave do formato obj

Após a cópia dos dados, deve-se garantir que eles serão desvinculados do *buffer* utilizando novamente a *glBindBuffer()*, mas dessa vez o parâmetro *id* como zero. Além disso, é necessário utilizar a chamada *glDeleteBuffers(GLsizei n, const GLuint * buffers)* (em que n é o número de objetos do *buffer* e *buffer* é o array de *buffers* a serem deletados) para poder liberar a memória. Os mesmos procedimentos podem ser feitos para criar o *buffer* de índices.

2.4.4 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 Tab. (3) 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 3 – 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 Fig. 8 mostra o exemplo de um arquivo obj para a leitura de um cubo.

```
# 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 8 – Arquivo obj 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.

2.5 Quadros Por Segundo

Frame rate é o quão rápido uma sequência de *frames*, ou quadros, são apresentados ao espectador. A unidade utilizada para determinar *frame rate* em jogos e filmes são os *frames per second* (FPS) ou quadros por segundo, que é o número de imagens renderizadas por segundo. O tempo usado por uma aplicação para gerar uma imagem varia dependendo da complexidade da computação desempenhada durante cada quadro. O FPS é utilizado tanto para expressar a taxa de um quadro em particular quanto para determinar o desempenho médio durante o uso da aplicação.

De acordo com (GREGORY, 2009), jogos na América do Norte e Japão são renderizados a 30 ou 60 quadros por segundo, porque essa é a taxa de atualização do sistema *National Television System Committee* (NTSC) usadas nessas regiões. Na Europa e no resto do mundo esta taxa é de 50 quadros por segundo, pois é a taxa de atualização dos televisores do tipo *Phase Alternating Line* (PAL) ou *Séquentiel Couleur à Mémoire* (SECAM). Todos estes sistemas são sistemas de televisores analógicos.

2.6 *Shaders*: *pipelines* programáveis

Um *shader* é responsável por determinar o efeito de uma luz em um material.

2.7 *Gouraud Shading*

2.8 *Phong Shading*

2.9 Ferramentas

2.9.1. *Adreno Profiler*

A *Adreno* é uma ferramenta que foca na otimização gráfica para celulares que possuem *Graphics processing unit* (GPU) Adreno (fabricada pela empresa *Qualcomm*). De acordo com (QUALCOMM,), 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.

Como pode ser visto na Fig. 9, a ferramenta possui um módulo de análise dos *vertex* e *fragment shaders*, sendo possível editá-los e analisar os resultados de compilação em tempo real, além dela também gerar estatísticas.

O módulo gráfico permite analisar algumas métricas, como a de quadros por segundo, em que na Fig. 10 um gráfico é plotado em tempo de execução. Além disso,

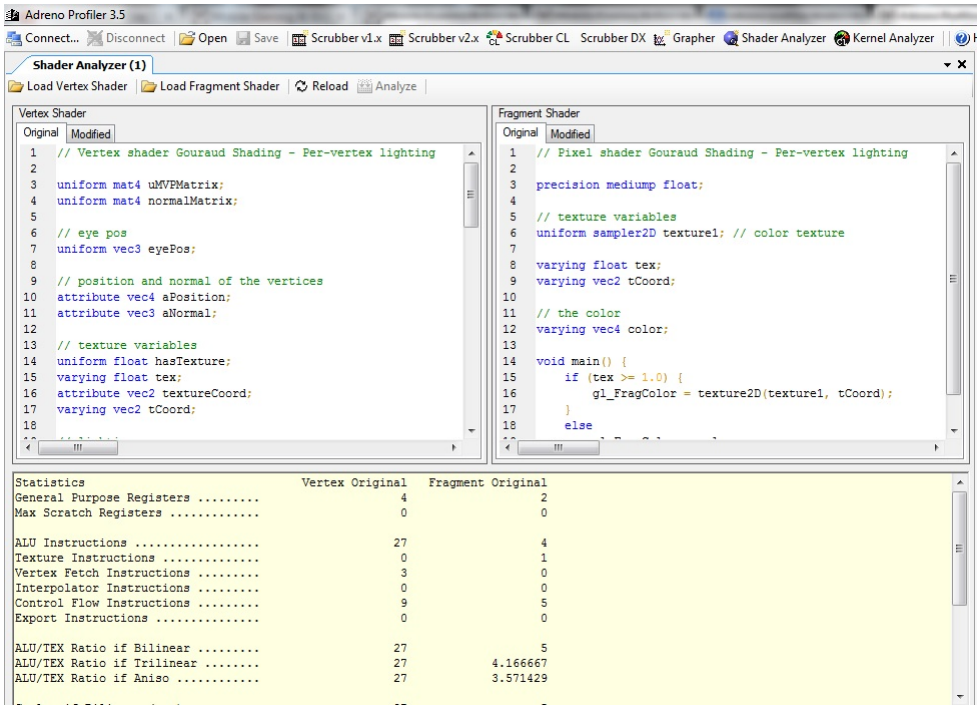

Figura 9 – Ferramenta *Adreno Profiler*: analisador de *shaders*

ela também exporta os resultados no formato *Comma-Separated Values* (CSV), que é 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.

Figura 10 – Ferramenta *Adreno Profiler*: quadros por segundo

2.9.2 *gDEBugger*

A *gDEBugger* é uma ferramenta de depuração e análise de desempenho (Fig. 11 e Fig. 12), que permite a rastreabilidade das chamadas *OpenGL* de uma aplicação,

disponível para *Windows* e *Linux*, com suporte às GPU's da empresa NVIDIA. Ela é responsável por testar outros programas que usam *OpenGL*, a fim de encontrar defeitos e possíveis otimizações e melhorar o desempenho.

Como é mostrado em (REMEDY,), é possível ver quais funções foram chamadas em um determinado quadro, o valor das variáveis da *OpenGL* (como as matrizes de projeção, visualização e modelagem, por exemplo) e também mostrar medições com relação a quadros por segundo, consumo de memória, número de função de chamadas por quadro, entre outros. Além disso, ela (assim como a ferramenta *Adreno Profiler*) também exporta os resultados no formato CSV.

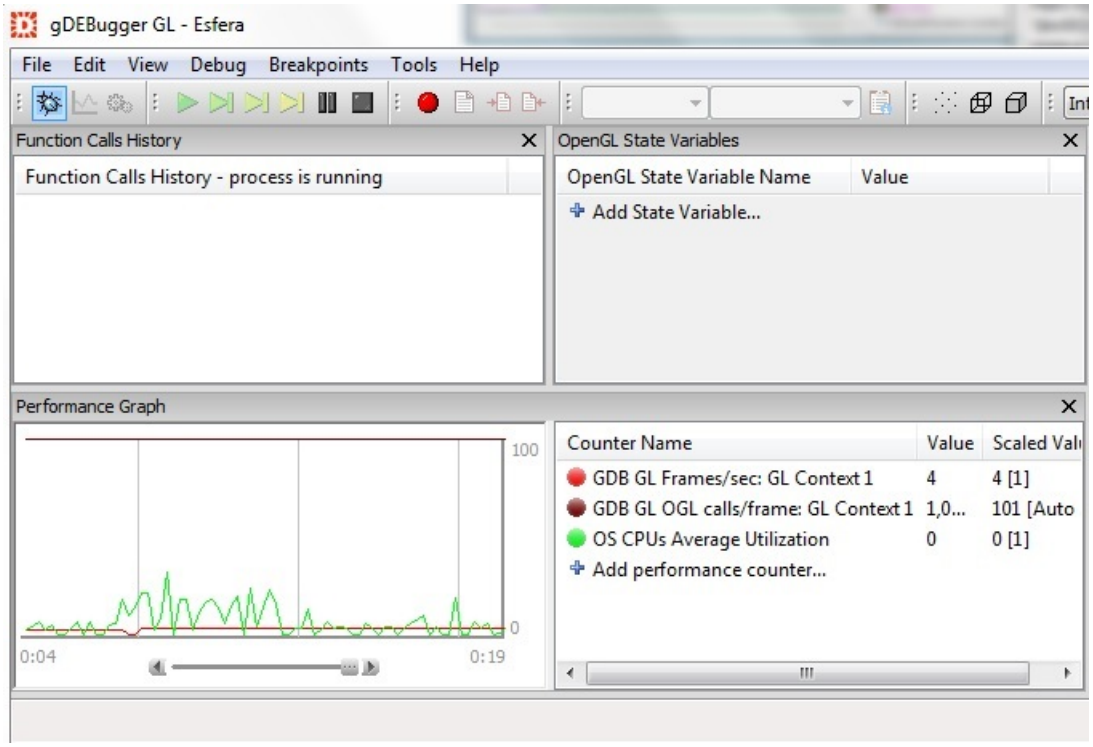

Figura 11 – Ferramenta *gDEBugger*: Gráfico de desempenho, histórico de chamadas e valor das variáveis

2.10 Complexidade Algorítmica

Complexidade algorítmica é uma medida que compara a eficiência de um determinado algorítmo, analisando o quão custoso ele é, e foi desenvolvida por Juris Hartmanis e Richard E. Stearns. 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 é relevante somente com relação a grandes quantidades de dados, os termos que não afetam a ordem de magnitude são eliminados e esta aproximação é

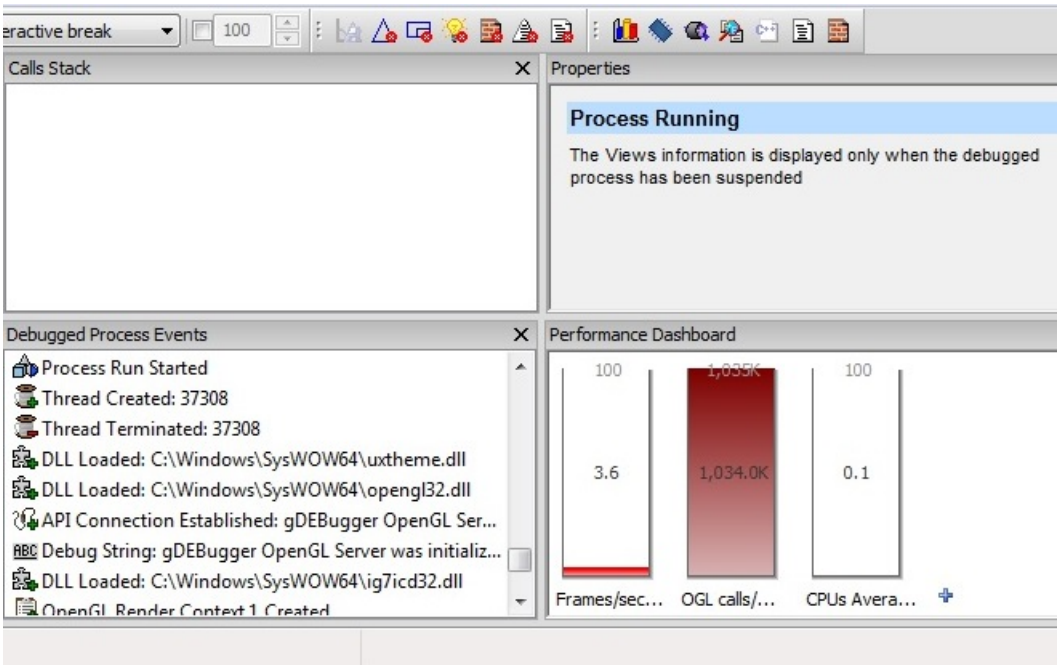

Figura 12 – Ferramenta *gDEBugger*: Medições de desempenho e eventos de depuração

denominada complexidade assintótica. Assim, a Eq. (2.1) poderia ser aproximada pela Eq. (2.2)

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

$$\mathcal{Y} \approx n^2 \tag{2.2}$$

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 Tab. (4) abaixo.

2.11 Métodos dos Mínimos Quadrados

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 4 – Valores mais comuns de complexidade algorítmica

3 Metodologia

3.1 Levantamento Bibliográfico

Sabendo do interesse, dentro da área de computação gráfica, especificamente no desenvolvimento de *shaders*, primeiramente foi feito um levantamento bibliográfico, a fim de avaliar a disponibilidade de material para fomentar o tema de trabalho de pesquisa 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, em que (como foi dito na seção 1 Introdução) viu-se que a limitação de desempenho e o desenvolvimento para *mobile* são áreas a serem exploradas.

3.2 Configuração do Ambiente

Assim, primeiramente foram feitas as configurações dos ambientes de trabalho, em que - como citado na Seção 2.10 - para desenvolver na plataforma *Android* é necessário instalar o *Android SDK* e o *plugin* ADT, este último a fim de poder desenvolver na IDE *Eclipse*. A biblioteca gráfica para sistemas embarcados *OpenGL ES* já é oferecida pela plataforma *Android*. Para computador, foi necessário instalar as bibliotecas *GLUT*, *GLEW* e por fim, a biblioteca gráfica *OpenGL*.

3.3 Equipamentos Utilizados

O celular utilizado foi o *Nexus* 4, no 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 *Adreno* 320 e 2 GB de memória RAM. O computador utilizado 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.

3.4 Definição do Tema

Primeiramente foi analisado se era factível estender o tema também para a plataforma *Android* - tanto no que diz respeito ao prazo quanto em relação também ao conhecimento já possuído. Então avaliou-se o o nível de dificuldade de implementação de um *shader* para plataforma *Android* (principalmente por não possuir experiência prévia com desenvolvimento *mobile*), desenvolvendo um *shader* simples aplicado num octaedro.

Também desenvolveu-se o mesmo *shader* para computador, analisando as as diferenças de implementação entre eles.

Feito isto, também foi realizado um levantamento de ferramentas de otimização gráfica tanto para *Android* como para computador, no qual escolheram-se as ferramentas *Adreno* e *gDEBugger*, respectivemente. Essas ferramentas são utilizadas a fim de coletar medições quanto ao número de quadros por segundo de cada programa, utilizando um *shader* específico, aplicado num objeto tridimensional com n número de polígonos.

A fim de facilitar a implementação dos *shaders*, também foi realizado um levantamento de ferramentas para o desenvolvimento de *shaders*, em que se escolheu a ferramenta *Render Monkey*.

Para poder finalmente verificar a viabilidade do tema em si, foi implementado um programa no qual é uma cena constituída por três esferas (com número de polígonos variável), em que cada uma faz uma movimentação diferente (em que garante-se que há oclusão e a distância em relação à câmera varia) e nas quais aplicam-se *shaders* específicos.

Assim, utilizando as ferramentas mencionadas anteriormente, foi possível coletar o número de quadros por segundo para diferentes números de polígonos. E dessa forma, gráficos (quadros por segundo x número de polígonos) para cada *shader* implementado foram traçados (Anexo I), podendo então analisar experimentalmente suas complexidades algorítmicas.

3.5 Procedimentos Futuros

Assim, os próximos passos estão relacionados com a escolha de quais *shaders* serão implementados e terão suas complexidades algorítmicas analisadas, como também com a modelagem de objetos tridimensionais possuindo diferentes números de polígonos e com a implementação do leitor desses objetos.

Por fim, o método dos mínimos quadrados será utilizado para poder estimar o número de quadros por segundo de um *shader*, dado um número n de polígonos, baseando-se na curva obtida experimentalmente pelos gráficos de cada *shader* tanto no computador quanto no celular.

4 Resultados Alcançados

4.1 Teste de Viabilidade do Tema

Para testar a viabilidade do tema proposto, a ideia foi criar um programa constituído por objetos que fossem fáceis de variar o número de polígonos. Esses objetos escolhidos foram esferas pela facilidade de implementação, pois já existe uma função pronta da biblioteca *glut* chamada *glutSolidSphere*, no qual se cria uma esfera baseada no tamanho do raio, número de cortes latitudinais e longitudinais. O número total de polígonos se dá pela multiplicação destes dois últimos parâmetros como é mostrado em (LINUX,). Além disso, estas esferas possuem diferentes movimentações, em que garante-se a ocorrência de oclusão entre elas e diferentes distâncias com relação à câmera. Feito isto, diferentes tipos de *shaders* foram aplicados nestas esferas - a fim de posteriormente fazer medições com relação aos quadros por segundo - e finalmente poder traçar gráficos entre quadros por segundo *versus* número de polígonos. Dessa forma é possível analisar a complexidade algorítmica experimentalmente. Devido ao prazo, este experimento foi feito somente no computador, principalmente pela *OpenGL ES* não possuir uma função equivalente à *glutSolidSphere* e precisaria de mais tempo para implementá-la.

5.1.1 *Shader* cor vermelha

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 (obtida utilizando o comando *gl-Vertex*) com a matriz de projeção,visualização e modelagem como é mostrada na Fig. 13.

```
// red.vs
//
// just multiplies vertex position vector by the modelview
// and projection matrices.

void main()
{
   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Figura 13 – *Red vertex shader*

Já o seu *fragment shader* estabele que todo fragmento possui a cor vermelha, como é mostrado na Fig. 14.

```
// red.fs
//
// Sets fragment color to red.

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

Figura 14 – *Red fragment shader*

O resultado da aplicação deste *shader* é mostrado na Fig. 15, em que a cor das esferas é vermelha e cada uma delas faz distintas movimentações.

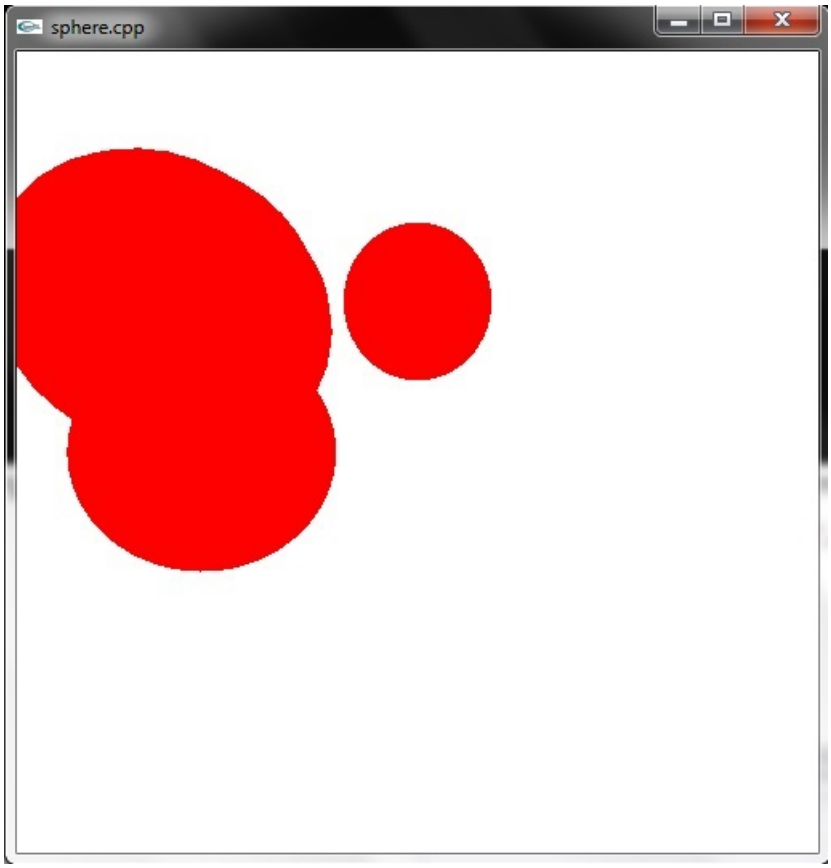

Figura 15 – *Red shader*

5.1.2 *Flatten shading*

A ideia do *flatten shading* é tornar o modelo tridimensional em bidimensional, achatado, e para isso, a coordenada z deve ser definida como zero. Mas para dar movimentação à malha do objeto, como é mostrado na Fig. 16, foi definida a variável *time* do tipo *uniform* que é inicializada e passada pelo programa para o *shader*. Assim, a coordenada z varia de acordo de acordo com o fator definido (que inclui esta variável).

```
uniform float time;

void main()
{

   vec4 v = vec4(gl_Vertex);
   v.z = sin(5.0*v.x + time*0.01)*0.25;
   gl_Position = gl_ModelViewProjectionMatrix * v;

}
```

Figura 16 – *Flatten vertex shader*

Neste caso o *fragment shader* não interfere no resultado desejado e por isso foi utilizado o mesmo do *red shader*, definindo a cor para vermelha. A Fig. 17 mostra as esferas achatadas, com a coordenada z variando de acordo com o fator definido.

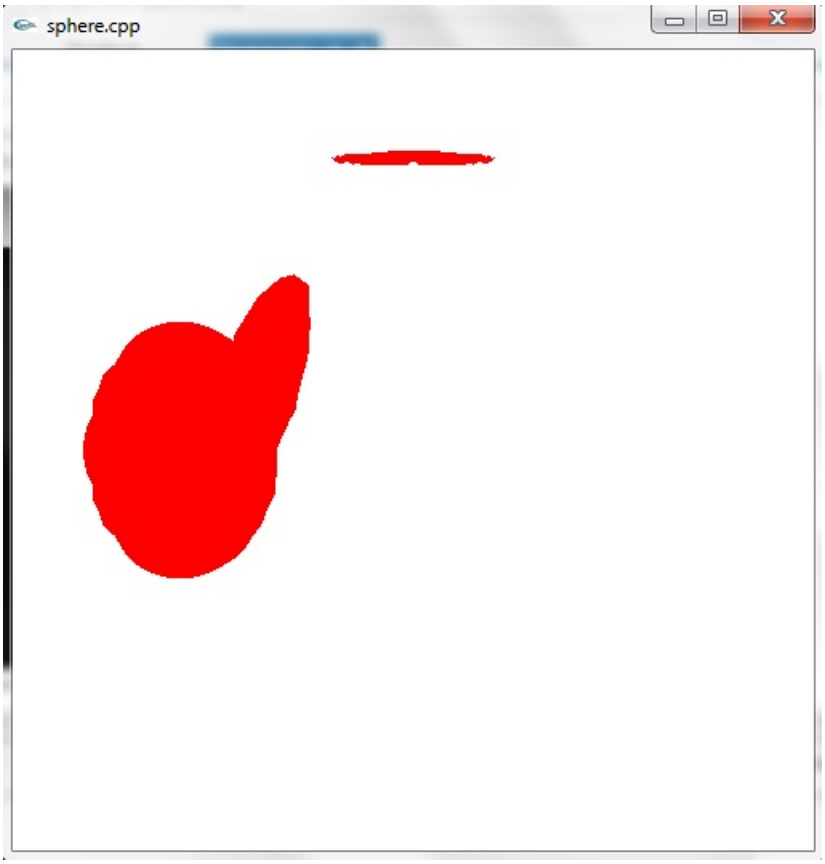

Figura 17 – *Flatten shader*

5.1.3 *Toon shading*

O *toon shading* calcula a intensidade da luz por vértice para escolher uma das quatro cores definidas. A Fig. 18 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 (adquirida através do comando *glNormal*).

```
uniform vec3 lightDir;

varying float intensity;

void main()
{
        vec3 ld;
        intensity = dot(lightDir,gl_Normal);
        gl_Position = ftransform();
}
```

Figura 18 – *Toon vertex shader*

A variável *intensity* do tipo *varying* é passada do *vertex shader* para o *fragment shader* - em que como mostra a Fig. 20 - para determinar qual das quatro cores será escolhida.

```
varying float intensity;

void main()
{
        vec4 color;

        if (intensity > 0.95)
                color = vec4(1.0,0.5,0.5,1.0);
        else if (intensity > 0.5)
                color = vec4(0.6,0.3,0.3,1.0);
        else if (intensity > 0.25)
                color = vec4(0.4,0.2,0.2,1.0);
        else
                color = vec4(0.2,0.1,0.1,1.0);

        gl_FragColor = color;
}
```

Figura 19 – *Toon fragment shader*

Assim, a direção da luz passada pelo programa é 0 , 1 , 1 e o resultado da aplicação do *shder* é mostrado na Fig. **??**.

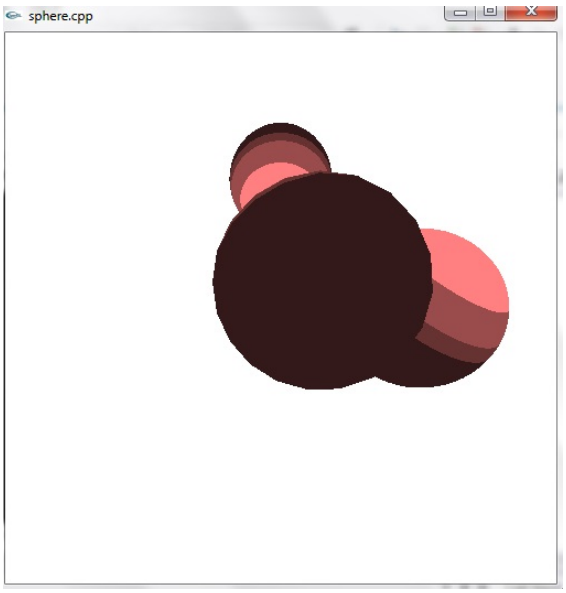

Figura 20 – *Toon shader*

5.1.4 *Phong shading*

O *vertex* e *fragment shaders* do *phong shading* implementam a técnica descrita na Seção 2.8. As Fig. 21 e Fig. 22 abaixo, mostram as definições do *vertex* e *fragment shaders*, respectivamente, em que para isso é necessário definir as propriedades do material pelo programa.

```
varying vec4 eyePosition;
varying vec3 diffuseColor;
varying vec3 specularColor;
varying vec3 emissiveColor;
varying vec3 ambientColor;
varying float shininess;
varying vec3 normal;

void main()
{

gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

eyePosition = gl_ModelViewMatrix * gl_Vertex;
normal = gl_NormalMatrix * gl_Normal;
diffuseColor = vec3(gl_FrontMaterial.diffuse);
specularColor = vec3(gl_FrontMaterial.specular);
emissiveColor = vec3(gl_FrontMaterial.emission);
ambientColor = vec3(gl_FrontMaterial.ambient);
 shininess = gl_FrontMaterial.shininess;
}
```

Figura 21 – *Phong vertex shader*

```
varying vec4 eyePosition;
varying vec3 normal;
varying vec3 diffuseColor;
varying vec3 specularColor;
varying vec3 emissiveColor;
varying vec3 ambientColor;
varying float shininess;
void main()
{
 const vec3 lightColor = vec3(1, 1, 1);
 const vec3 globalAmbient = vec3(0.2, 0.2, 0.2);
 vec3 P = vec3(eyePosition);
 vec3 N = normalize(normal);
 vec3 emissive = emissiveColor;
 vec3 ambient = ambientColor * globalAmbient;
 vec3 L = normalize(vec3(gl_LightSource[0].position) - P);
 float diffuseLight = max(dot(N, L), 0);
 vec3 diffuse = diffuseColor * lightColor * diffuseLight;

// Compute the specular term
 vec3 V = normalize(-P);
 vec3 H = normalize(L + V);
 float specularLight = pow(max(dot(N, H),0), shininess);
 if(diffuseLight <= 0) specularLight = 0;
 vec3 specular = specularColor * lightColor * specularLight;
 gl_FragColor.xyz = emissive + ambient + diffuse + specular;
 gl_FragColor.w = 1.0;
}
```

Figura 22 – *Phong fragment shader*

O resultado é mostrado na Fig. 23, no qual é muito parecido com o resultado padrão implementado pela *OpenGL*.

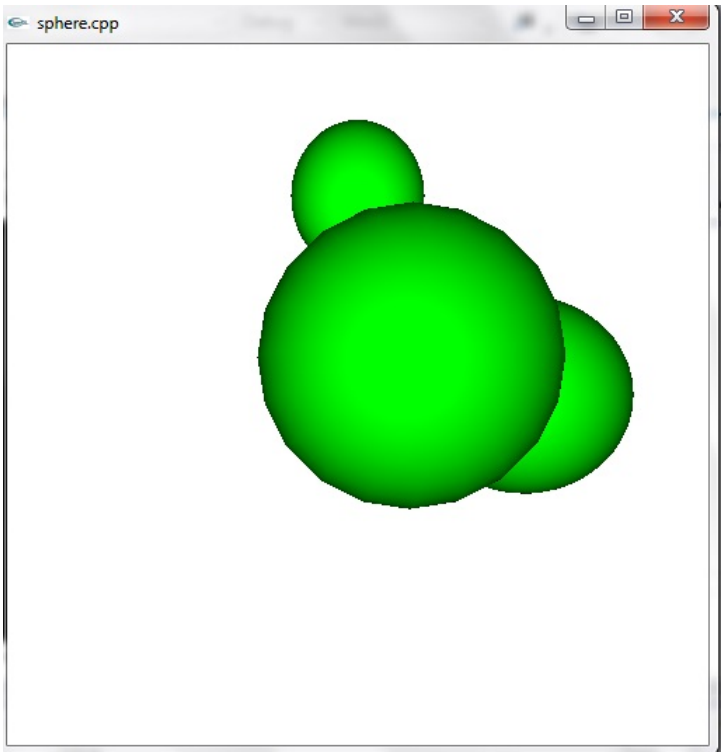

Figura 23 – *Phong shader*

5.1.5 *Texture shading*

O *vertex shader* do *texture shading* primeiramente armazena, numa variável do tipo *varying*, as coordenadas da textura por meio do comando *glMultiTexCoord0* para repassar para o *fragment shader*. Vale ressaltar que para este *shader* foi necessário utilizar a função *gluSphere* ao invés da *glutSolidSphere*, pois ela permite especificar um objeto do tipo quádrica, que por sua vez dá opção de criar coordenadas de textura para quádricas (utilizadas pelo *shader*).

```
varying vec2  TexCoord;

void main(void)
{
 TexCoord = gl_MultiTexCoord0;
 gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Figura 24 – *Texture vertex shader*

O *fragment shader* por sua vez, utiliza a textura da Fig. 25 passada pelo programa e aplica na coordenada repassada pelo *vertex shader*, como mostra a Fig. 26. Assim, o resultado é mostrado na Fig. 27.

Figura 25 – Textura utilizada

```
uniform sampler2D myTexture;
varying vec2 TexCoord;

void main (void)
{
  gl_FragColor  = texture2D(myTexture, TexCoord);
}
```

Figura 26 – *Fragment shader*

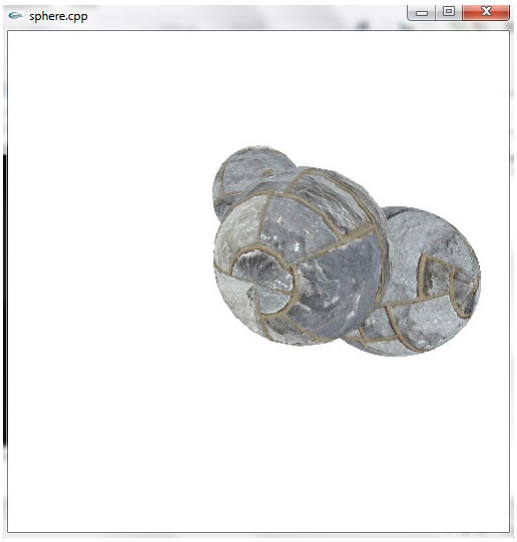

Figura 27 – *Texture shader*

4.2 Gráficos e Análise de Complexidade Algorítmica

Após a implementação dos *shaders*, foi utilizada a ferramenta *gDEBugger* descrita na Seção 2.9.2 para fazer medições quanto ao número de quadros por segundo, medida escolhida a fim de avaliar o desempenho, para n números de polígonos. Essa medida de desempenho foi adotada para poder avaliar experimentalmente as complexidades algorít-

micas dos *shaders* através das curvas plotadas.

A Fig. 28 mostra essa ferramenta sendo utilizada, na qual executa-se o programa e a métrica desejada é mostrada em tempo de execução e também pode ser pode ser exportada no formato *Comma-Separated Values*.

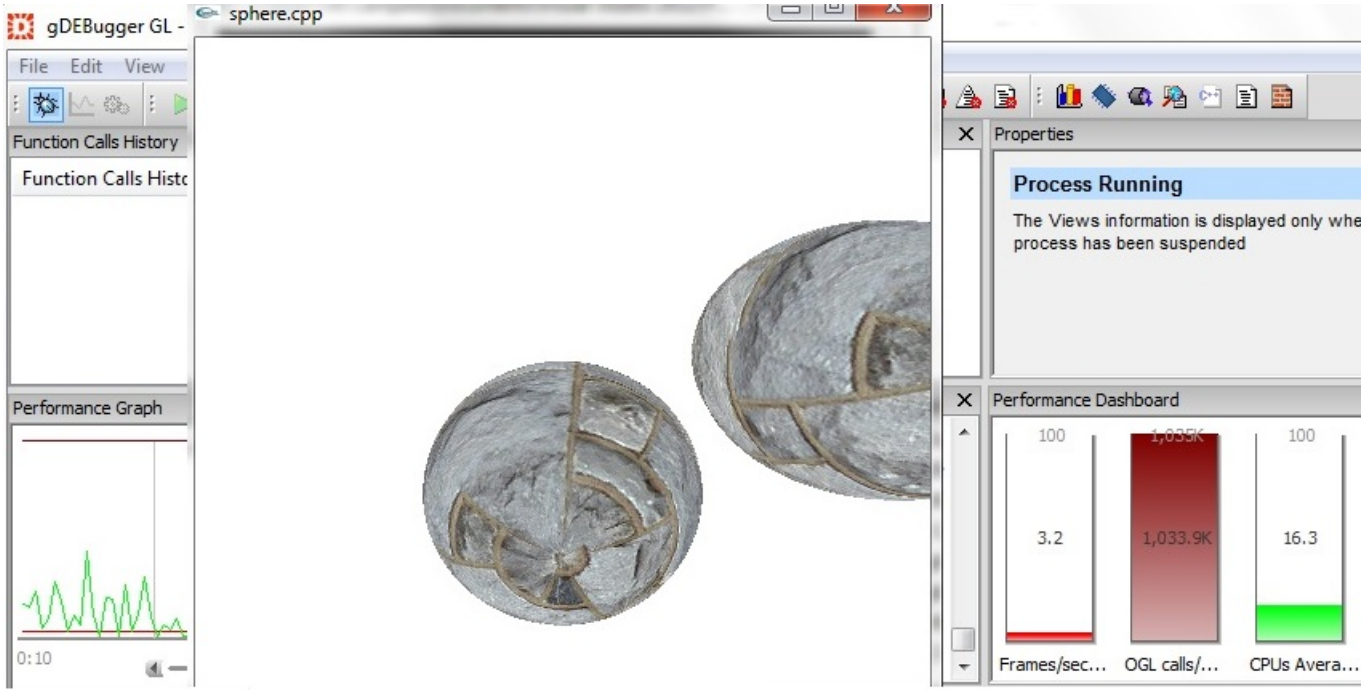

Figura 28 – Ferramenta *gDEBugger* sendo utilizada

Para cada número de subdivisões (no qual foi utilizado o mesmo valor para as subdivisões de latitude e longitude) coletaram-se dez medições, fazendo-se então a média aritmética. O número total de polígonos é dado pela Eq. (4.1), em que y é o número de polígonos, sub é o número de subdivisões e esf é a quantidade de esferas. Ao final multiplica-se por dois, pois as subdiviões geram quadriláteros e multiplicando por dois contabilizam-se o número de triângulos (que é o desejado).

$$\mathcal{Y} = sub^2 \cdot esf \cdot 2 \tag{4.1}$$

Assim, as medições foram realizadas, para cada *shader* implementado (*Red, Flatten, Toon, Phong* e *Texture*). Além disso, variou-se o número de subdivisões das esferas, começando a partir de 50 e incrementando de 25 em 25 até o máximo de subdivisões possíveis, que no caso da esfera utilizada de raio 10, foi de 250. Após este valor, não é possível subdividir mais utilizando as funções da bilblioteca *Glut*. As Tab. (5), Tab. (6), Tab. (7), Tab. (**??**), Tab. (**??**) abaixo mostram essas medições, evidenciando a quantidade de quadros pro segundo para determinado número de polígonos. Estas tabelas foram criadas a fim de poder, posteriormente, plotar os gráficos desejados para a análise de complexidade.

Nº de subdivisões	**Nº de polígonos**	**Quadros/Segundo**	**Quadros/Segundo**
50	15000	95.2	95.4
75	33750	49.6	47.1
100	60000	28.6	26.3
125	93750	18.8	18.3
150	135000	13.1	12.7
175	183750	9.6	9.4
200	240000	7.4	7.1
225	303750	6.0	5.9
250	375000	5.2	4.9

Tabela 5 – *Red Shader* e *Flatten Shader*, respectivamente

Nº de subdivisões	**Nº de polígonos**	**Quadros/Segundo**	**Quadros/Segundo**
50	15000	95.2	96.6
75	33750	48.8	49.3
100	60000	28.3	28.6
125	93750	18.1	18.4
150	135000	12.9	13.1
175	183750	9.6	9.7
200	240000	7.5	7.4
225	303750	5.9	5.8
250	375000	5.2	5.2

Tabela 6 – *Toon Shader* e *Phong Shader*, respectivamente

Nº de subdivisões	**Nº de polígonos**	**Quadros/Segundo**
50	15000	76.9
75	33750	33.4
100	60000	20.3
125	93750	13.1
150	135000	9.2
175	183750	6.8
200	240000	5.2
225	303750	4.1
250	375000	3.4

Tabela 7 – *Texture Shader*

Com estes dados foi possível traçar as curvas de complexidade algorítmica relativas a cada *texture shader* implementado, tomando como base a quantidade de quadros por segundo. Assim, todas elas foram plotadas em um único gráfico, como é possível ver na Fig. 29, aparentam a forma de uma função exponencial decrescente. E com exceção da curva do *texture shader* - que requer maior poder de processamento - as curvas dos outros *shaders* ficaram muito próximas, tanto que no gráfico algumas delas não é possível

enxergar.

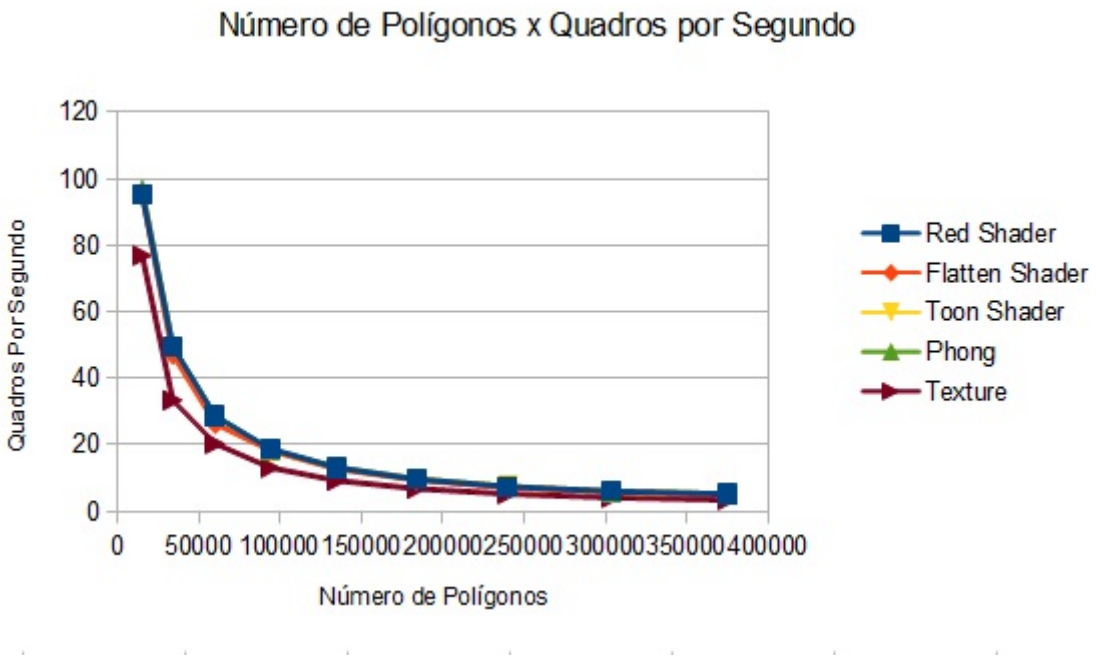

Figura 29 – Complexidade algoritma: exponencial

De acordo com (REFERENCIAR), a função da exponencial pode ser dada como na Eq. (4.2), em que e, c, k são constantes (e é a constante neperiana).

$$\mathcal{Y} = ce^{-kt} \tag{4.2}$$

Aplicando a função logarítimo dos dois lados da equação, obtém-se a Eq. (4.3).

$$\ln \mathcal{Y} = \ln c + \ln e^{-kt} \tag{4.3}$$

Que pode ser simplificada na Eq. (4.4), em que b é uma nova constante, e de acordo com (REFERENCIAR) equivale à equação da reta.

$$\mathcal{Y} = b - kt \tag{4.4}$$

Assim, para confirmar experimentalmente se a curva obtida dos gráficos é realmente uma exponencial, traçaram-se os gráficos novamente na Fig. 30, mas dessa vez, na escala logarítmica.

Analisando-se os gráficos, é possível perceber que todos eles se assemelham à uma reta, confirmando as suspeitas levantadas de que a complexidade algorítmica é na ordem exponencial.

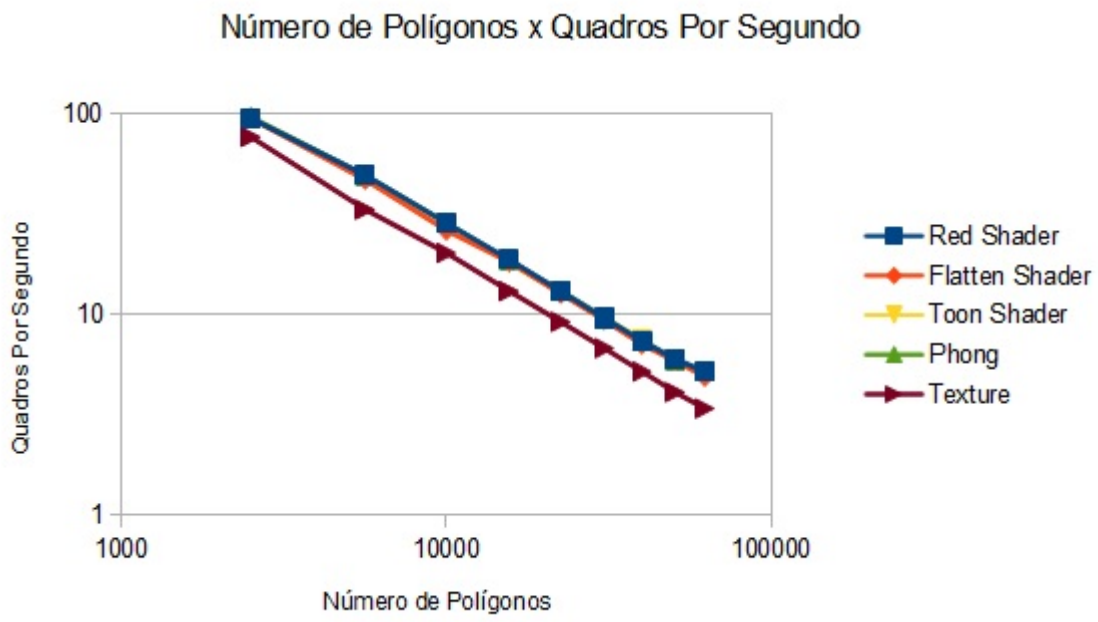

Figura 30 – *Complexidade Algorítmica: reta*

4.3 Implementação Plataforma *Android*

Para analisar a dificuldade de se implementar na plataforma *Android*, codificou-se o *Gouraud shader* descrito na Seção 2.7, que utiliza a biblioteca *OpenGL ES* e a linguagem de programação Java. Então o programa lê um arquivo obj (descrito na Seção 2.4.4) e renderiza o modelo tridimensional utilizando a técnica de *Vertex Buffer Object* descrita na Seção 2.2.3 e o *shader* mencionado. O resultado se encontra na Fig. 31.

Figura 31 – Implementação plataforma *Android*

Assim, verificou-se que é possível implementar para a plataforma *Android* dentro do prazo estipulado.

4.4 Implementação Computador

A fim de se ter uma comparação com o computador, já que a complexidade algorítmica não muda, codificou-se o mesmo programa mostrado na Fig. 32 , porém utilizando a biblioteca OpenGL, Glut e a linguagem de programa C++.

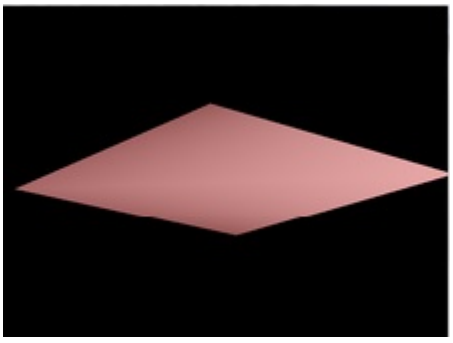

Figura 32 – Implementação Computador

4.5 Conclusão

Através dos experimentos realizados foi possível verificar que é factível desenvolver *shaders* para a plataforma *Android* dentro do prazo estipulado. E por meio da plotagem dos gráficos, a ideia de analisar a complexidade algorítmica experimentalmente foi plausível, já que foi obtida uma curva exponencial. Assim, futuramente mais *shaders* poderão ser analisados e implementados tanto para computador quanto para *Android*, utilizando modelos tridimensionais mais complexos, por meio da leitura do arquivo obj. E além disso, com base nas equações extraídas das curvas, o método dos mínimos quadrados - descrito na Seção 2.11 - será utilizado para estimar a quantidade de quadros por segundo dado um número de polígonos.

Outro ponto importante a que chegou-se foi com relação entre as diferenças da *OpenGL* e a *OpenGL ES*, enquanto convertia-se o mesmo programa para computador. O principal aspecto notado foi com relação à remoção das chamadas *glBegin* e *glEnd* para desenhar primitivas gráficas e a não utilização do *pipeline* convencional a partir da versão 2.0, em que faz-se obrigatória a utilização de *shaders* pela *OpenGL ES*. Além disso, todo o controle de matrizes de projeção, modelagem (operações de translação, rotação e escalar, por exemplo) e visualização fica a cargo do programador e não mais da *OpenGL ES*.

5 Cronograma de Desenvolvimento

Referências

ANGEL, E.; SHREINER, D. Interactive computer graphics. a top-down approach with shader-based opengl. Boston, Massachusetts, p. 689–690, 2012. Citado na página 24.

ANGEL, E.; SHREINER, D. Pro opengl es for android. New York, New York, p. 114–117, 2012. Citado na página 19.

BROTHALER, K. Opengl es 2 for android. Dallas, Texas, p. 234–237, 2013. Citado na página 24.

DROZDEK, A. Estrutura de dados e algoritmos em c++. São Paulo, São Paulo, p. 48–52, 2002. Citado na página 28.

GREGORY, J. Game architecture. Boca Raton, Florida, p. 312–316, 2009. Citado na página 26.

GUHA, S. Computer graphics through opengl: From theory to experience. Boca Raton, Florida, p. 114–117, 2011. Citado 3 vezes nas páginas 19, 22 e 23.

JACKSON, W. Learn android app development. New York, New York, p. 1–13, 2013. Citado na página 17.

LINUX, M. P. *glutSolidSphere*. Disponível em: < http://www.pkill.info/linux/man/3-glutSolidSphere/ >. Acessado em: 30 out. 2013. Citado na página 33.

MOLLER, T. A.; HAINES, E.; HOFFMAN, N. Real-time rendering. Boca Raton, Florida, p. 114–117,13–25, 2008. Citado na página 19.

QUALCOMM, D. N. *Mobile Gamimg and Graphics Optimization (Adreno) Tools and Resources*. 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 26.

REMEDY, G. *Save Precious Debugging Time and Boost Application Performance*. Disponível em: < http://www.gremedy.com/ >. Acessado em: 15 out. 2013. Citado na página 28.

SANDBERG, R.; ROLLINS, M. The business of android apps development. New York, New York, p. 4–10, 2013. Citado na página 17.

SEDGEWICK, R. Algorithms in c. Westford,Massachusetts São Paulo, p. 67–75, 1990. Citado na página 29.

SHERROD, A. Game graphics programming. Boston, Massachusetts, p. 600–609, 2011. Citado na página 24.

WRIGHT, R. S. et al. Opengl superbible: Comprehensive tutorial and reference. Boston, Massachusetts, p. 114–117, 2008. Citado 2 vezes nas páginas 18 e 19.

Anexos

ANEXO A – Primeiro Anexo

Texto do primeiro anexo.

ANEXO B – Segundo Anexo

Texto do segundo anexo.