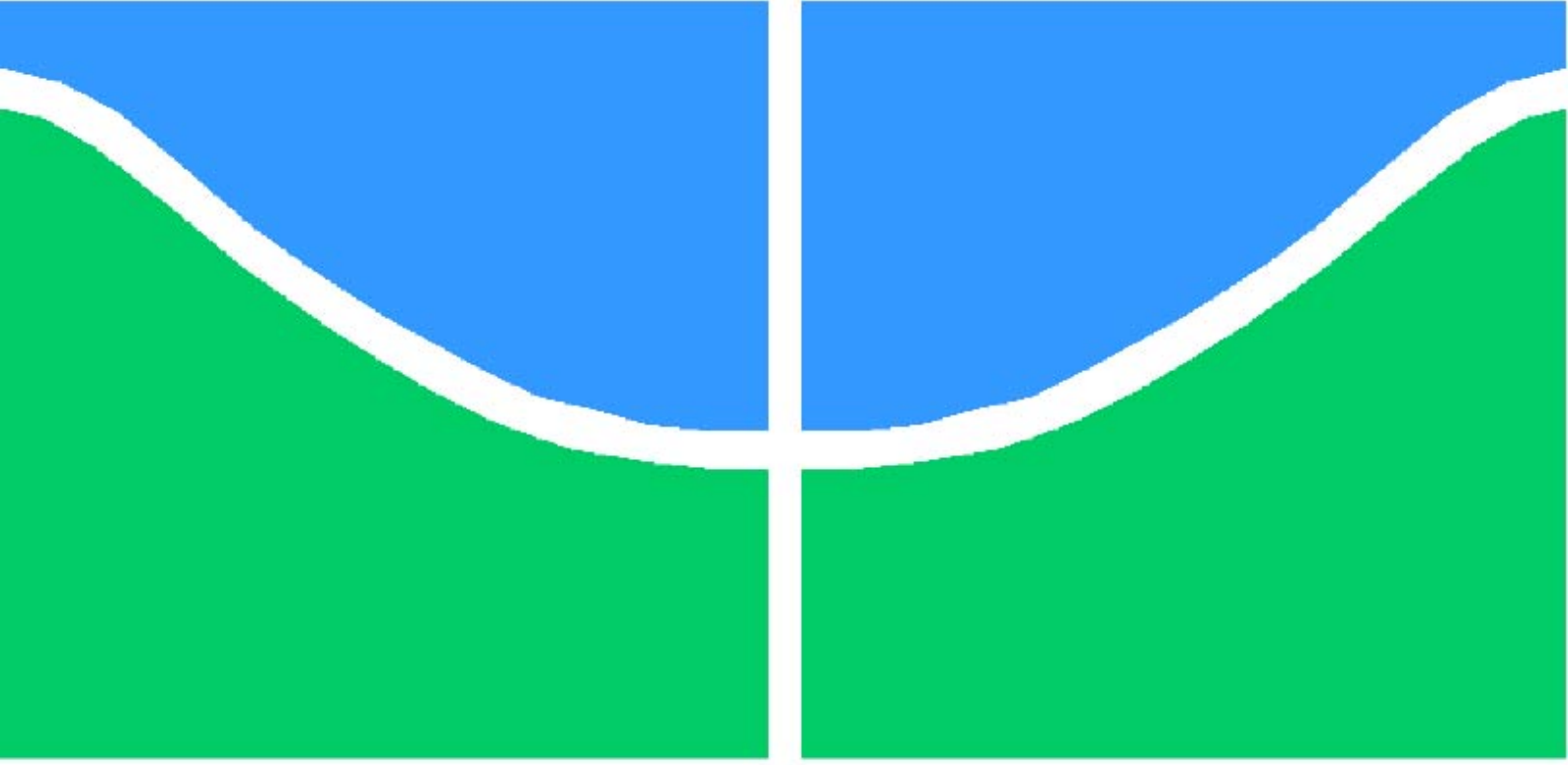

Universidade de Brasília - UnB

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*

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*

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 (discutido na Seção 5).

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 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 1 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	*Jelly Bean*
4.4	*KitKat*

Tabela 1 – Versões da plataforma *Android*

Uma das alternativas para o desenvolvimento em plataforma *Android* é utilizar a ferramenta *Eclipse*[1] (Figura 1), 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 *plu-*

[1] http://www.eclipse.org/
[2] http://developer.android.com/sdk/index.html

gin 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.

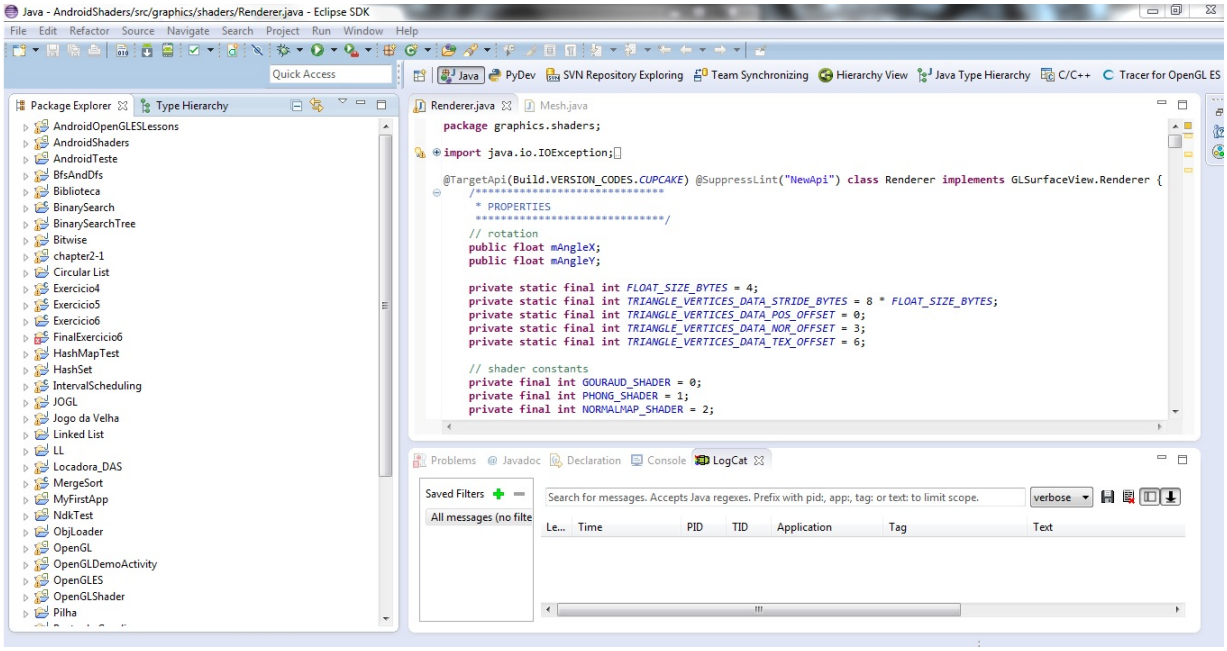

Figura 1 – Ambiente de desenvolvimento *Eclipse*

2.2 Bibliotecas Gráficas

2.2.1 *OpenGL*

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

Ela possui comandos para desenho de primitivas geométricas (como linhas e pontos, por exemplo), texturização, transparência, animação, entre outros efeitos especiais. Porém, ela não possui funções de gerenciamento de janela, eventos de entrada (*mouse* e teclado, por exemplo) ou leitura e escrita de arquivos: o próprio programador é respon-

[3] http://developer.android.com/tools/sdk/eclipse-adt.html
[4] http://developer.android.com/sdk/installing/studio.html

sável por configurar o ambiente necessário para a *OpenGL* desenhar em uma janela (seja para *Microsoft Windows*, *MacOS* ou *Unix*, por exemplo).

A *OpenGL* possui quatro versões, sendo a mais atual a 4.4. As principais modificações ocorreram entre as versões 1.x e 2.x – que permitiram o uso de *shaders* e *pipeline* de renderização programável – e entre as versões 2.x e 3.x, que depreciaram as funções fixas (que são removidas nas futuras versões da API).

2.2.2 *Glut*

Visando a portabilidade e abstração do sistema operacional, a biblioteca Glut (*OpenGL Utility Toolkit*) foi criada por Mark Kilgard, enquanto ele ainda trabalhava na empresa *Silicon Graphics*. Ela facilita a utilização de janelas e o tratamento de eventos de entrada, integrando as janelas do sistema operacional subjacente com a *OpenGL* de forma portável. Embora possua limitações relativas à interface gráfica com o usuário (*Graphical User Interface* - GUI), de acordo com (WRIGHT et al., 2008), ela é simples de ser utilizada.

2.2.3 *OpenGL ES*

A *OpenGL ES* (*OpenGL for Embedded Systems*) 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 (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 e a 3.x, que é completamente compatível com a *OpenGL* 4.3.

2.3 Processo do *Rendering Pipeline*

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 pode ser divido em dois subprocessos principais: o de geometria e o de rasterização. Segundo (MOLLER; HAINES; HOFFMAN, 2008), o processo de geometria pode ser divido nas etapas mostradas na Figura 2.

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

Figura 2 – Etapas do subprocesso de geometria

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 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 as características dos materiais atribuídos aos objetos, como também os efeitos da luz, sendo que cada efeito pode ser modelados 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. Assim, os resultados do *vertex shading* são mandados para o subprocesso de rasterização para serem interpolados.

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.

Somente as primitivas gráficas que se encontram dentro do volume de visualização que 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 3 mostra esta ideia.

A última etapa de geometria é a de Mapeamento na Tela, 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 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 nas quatro etapas mostradas na Figura 4.

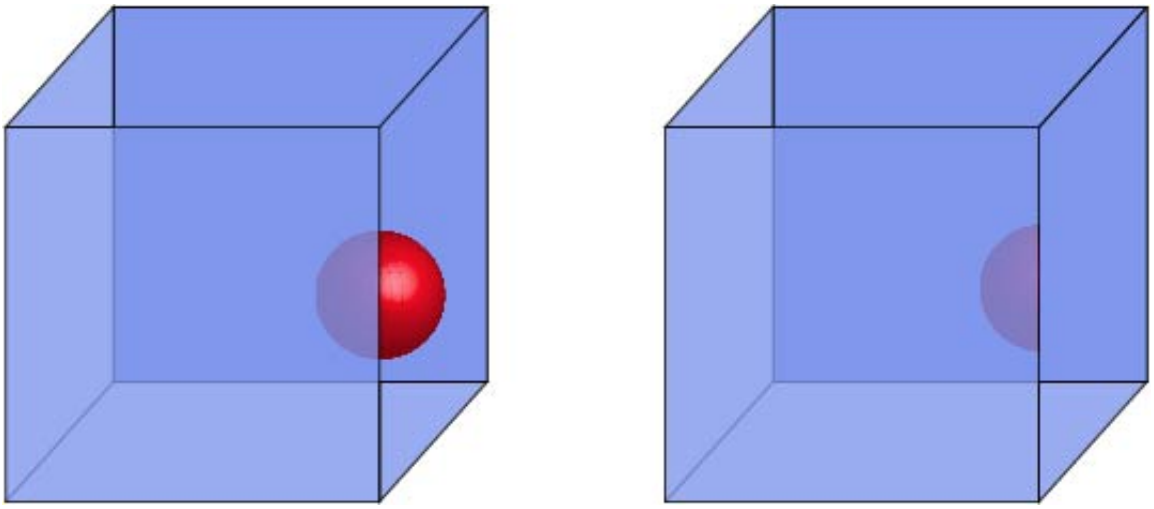

Figura 3 – Antes do recorte (cubo de visualização esquerdo) e depois do recorte (cubo de visualização direito)

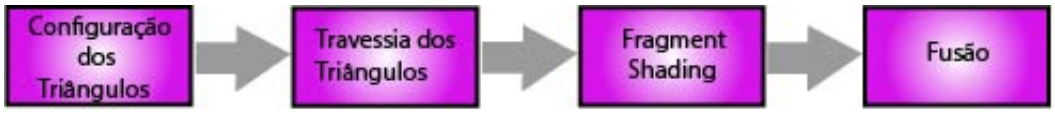

Figura 4 – Etapas do processo de rasterização

Na etapa de Configuração dos Triângulos, 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.

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, conforme mostrado na Figura 5. 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.

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 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

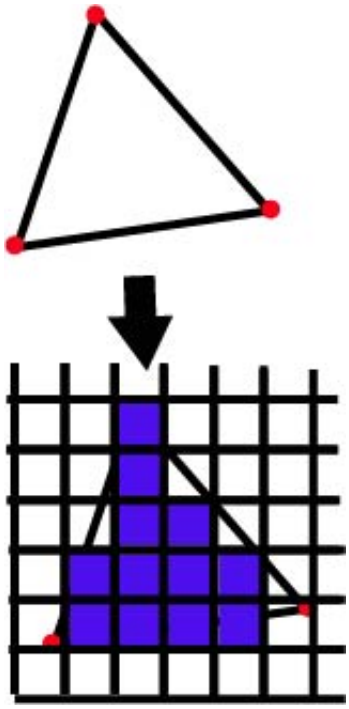

Figura 5 – Travessia de triângulos: fragmentos sendo gerados

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 Métodos de Renderização de Modelos Tridimensionais

2.4.1 Funções Fixas

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. As vantagens das funções fixas é que elas são mais convenientes: assim, um programador com pouca experiência em *OpenGL* ou *OpenGL ES* pode ter uma aprendizagem mais rápida. Em contrapartida, este método não dá flexibilidade ao programador de modificar algumas das etapas de renderização (vistas na Seção 2.3).

Uma das operações fixas é 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()`) para se desenhar um objeto. O parâmetro `Mode` diz qual primitiva deve ser utilizada, podendo ser: pontos; linhas; série de linhas conectadas; série de linhas conectadas em *loop* (em que conectam-se também o primeiro e último vértices); triângulos; série de triângulos conectados; quadriláteros; série de quadriláteros e polígonos convexos

simples.

Para se definir a cor, utiliza-se o comando `glColor3f(float r, float g, float b)`, em que os argumentos são as coordenadas do modelo RGB (*Red, Green and Blue*) e neste modelo, define-se cada cor pela quantidade de vermelho, verde e azul que a compõe. Um argumento a mais pode ser adicionado (utilizando-se o comando `glColor4f(float r, float g, float b, float alpha)`), em que o argumento *alpha* é o valor de transparência (o valor zero indica transparência, 1 (um) opacidade e os valores intermediários as diferentes gradações de translucidez).

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 na Figura 6 e na Figura 7.

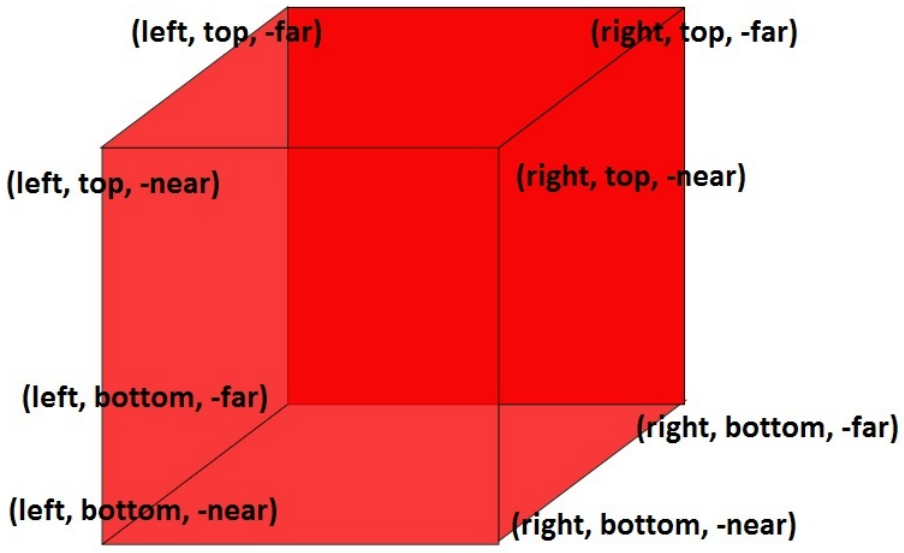

Figura 6 – Projeção Ortográfica: parâmetros

2.4.2 *Vertex Array*

Um modelo tridimensional pode ser descrito por uma lista de vértices e uma lista de índices. Na Figura 8, tem-se dois triângulos e quatro vértices definidos (dois vértices

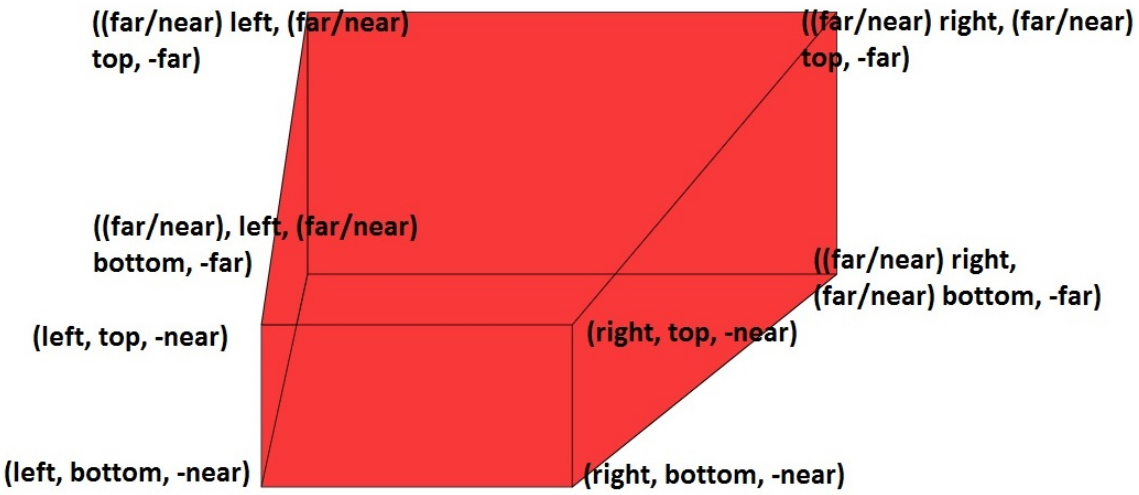

Figura 7 – Perspectiva: parâmetros

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.

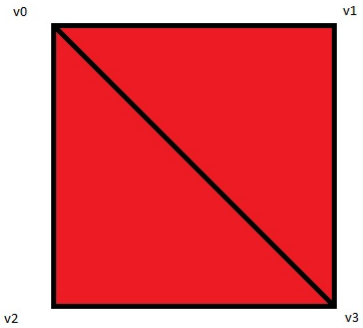

Figura 8 – Vértices do quadrado constituído de dois triângulos

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(G_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 (pon-

tos, 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 permitindo que diversos dados (como coordenadas de posição, textura, normais, por exemplo) possam ser definidos em um único vetor.

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, 2012), é necessário utilizar a chamada`glGenBuffers()`. Feito isto, vincula-se o vetor ao *buffer* com a função `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)`: `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 Tabela 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 – Padrões de uso de dados do `vertex buffer object`

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 `buffers` é o array a ser deletado) 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 Tabela 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 Figura 9 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.

2.5 Fundamentos de Animação

Uma animação é formada por uma coleção de quadros que são vistos um atrás do outro dentro de um intervalo de tempo, em que cada quadro é um instantâneo diferente da animação que está sendo realizada. Assim, *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 é o FPS (*Frames Per Second*) 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

```
#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 9 – Arquivo *obj* de um cubo

NTSC (*National Television System Committee*) 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 PAL (*Phase Alternating Line*) ou SECAM (*Séquentiel Couleur à Mémoire*). Todos estes sistemas são sistemas de televisores analógicos. Porém, a partir de 12 quadros por segundo já é possível provocar uma ilusão de que algo está se movendo continuamente.

2.6 *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. Então, *shaders* são algorítmos escritos pelo programador a fim de substituir as funcionalidades pré-definidas. Existem dois tipos de *shader*, que focam diferentes partes do *pipeline* gráfico: o *vertex shader* e o *fragment shader*.

O *vertex shader* é responsável pela manipulação dos dados dos vértices, incluindo coordenadas, normais, cores, sendo responsável pela alteração de posição e textura, por exemplo. Ele altera a etapa de *vertex shading*, descrita na Seção 2.3. Ele deve, ao menos,

definir as coordenadas de posição.

O *fragment shader* opera nos fragmentos no processo de rasterização (selecionar e colorir os *pixels*) antes de passar para a etapa de Fusão descrita na Seção 2.3, que faz as operações por fragmento (como o teste de profundidade). Ele deve, ao menos, atribuir uma cor para cada fragmento.

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 (através dos *shaders*), substituindo as funções fixas. 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 4.

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 4 – GLSL: tipos de dados

Além disso, a GLSL possui variáveis chamadas qualificadoras, que fazem o interfaceamento do programa e os *shaders* e entre *shaders*. Estas varáveis são mostradas na Tabela 5.

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 5 – GLSL: qualificadores

2.7 *Flat Shading*, *Gouraud Shading* e *Phong Shading*

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 10 mostra a diferença entre as três técnicas de *shading* aplicadas em uma esfera com uma luz direcional.

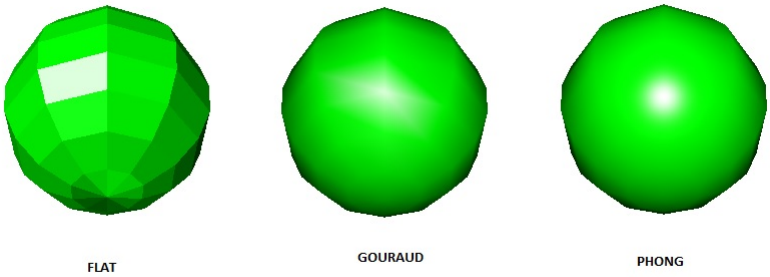

Figura 10 – Comparação entre as técnicas de *shading*

2.8 Complexidade Algorítmica

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 6.

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 6 – Valores mais comuns de complexidade algorítmica

2.9 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:

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

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

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

3 Metodologia

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 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, identificando (como foi dito na Seção 1 1) a limitação de desempenho e o desenvolvimento para *mobile* como áreas a serem exploradas.

3.2 Equipamentos Utilizados

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 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, em que para desenvolver na plataforma *Android* é necessário instalar o *Android SDK* e o *plugin* ADT, uma vez que seria utilizada a 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*. Para poder realizar a coleta de métricas, foram utilizadas as ferramentas (descritas nas subseções a seguir) *Adreno Profiler*, pois o celular utilizado possui a GPU *Adreno*, e a *gDEBugger*, uma das únicas ferramentas gratuitas que suporta GPU's da empresa NVIDIA.

3.3.1 *Adreno Profiler*

A *Adreno* 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 ope-

racional *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 Figura 11, 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.

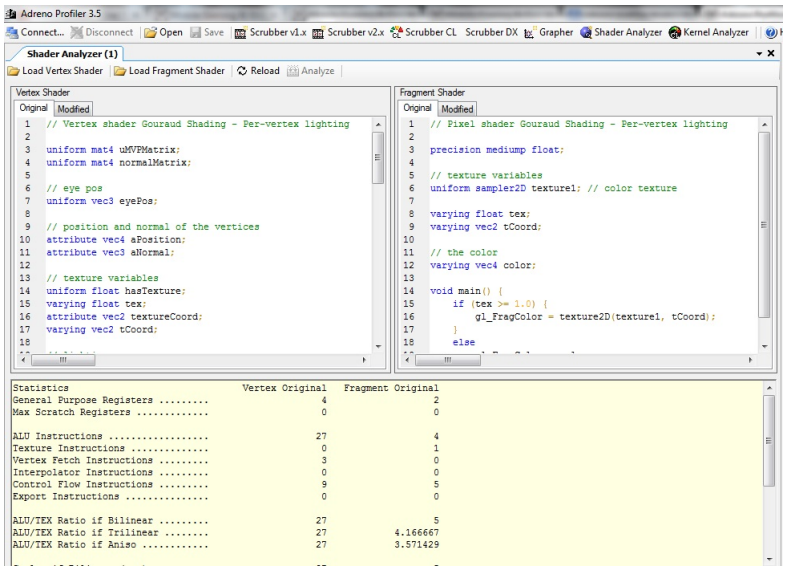

Figura 11 – Ferramenta *Adreno Profiler*: analisador de *shaders*

O módulo gráfico permite analisar algumas métricas, como a de quadros por segundo, onde na Figura 12 um gráfico é plotado em tempo de execução. Além disso, ela também exporta os resultados no formato CSV (*Comma-Separated Values*), que consiste em um arquivo de texto que armazena valores tabelados separados por um delimitador (vírgula ou quebra de linha). O último módulo é o chamado *Scrubber*, que provê informações detalhadas quanto ao rastreamento de uma chamada de função.

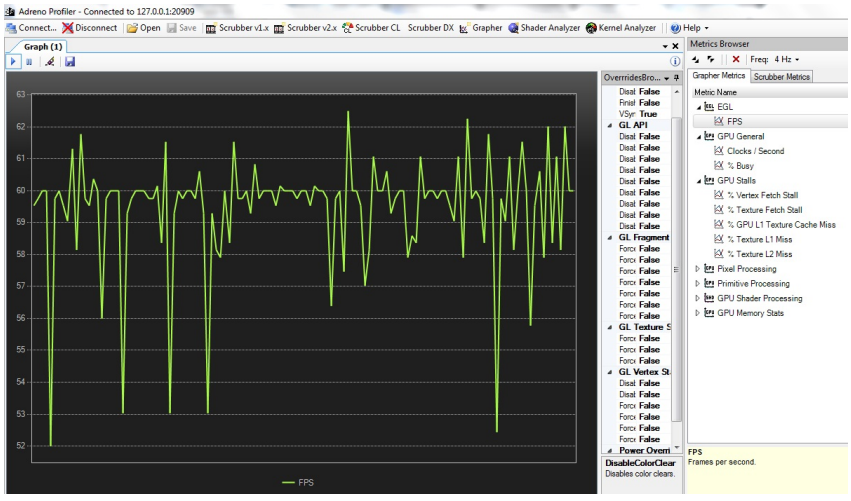

Figura 12 – Ferramenta *Adreno Profiler*: visualização de métrica quadros por segundo

3.3.2 *gDEBugger*

A *gDEBugger* é uma ferramenta de depuração e análise de desempenho (Figura 13), que permite a rastreabilidade das chamadas *OpenGL* de uma aplicação, disponível para *Windows* e *Linux*, com suporte às GPU's da empresa NVIDIA. Ela ajuda a encontrar *bugs*, melhorar o desempenho e consumo de memória de programas que utilizam a *OpenGL*.

Como é mostrado em (GASTER et al., 2013), 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.

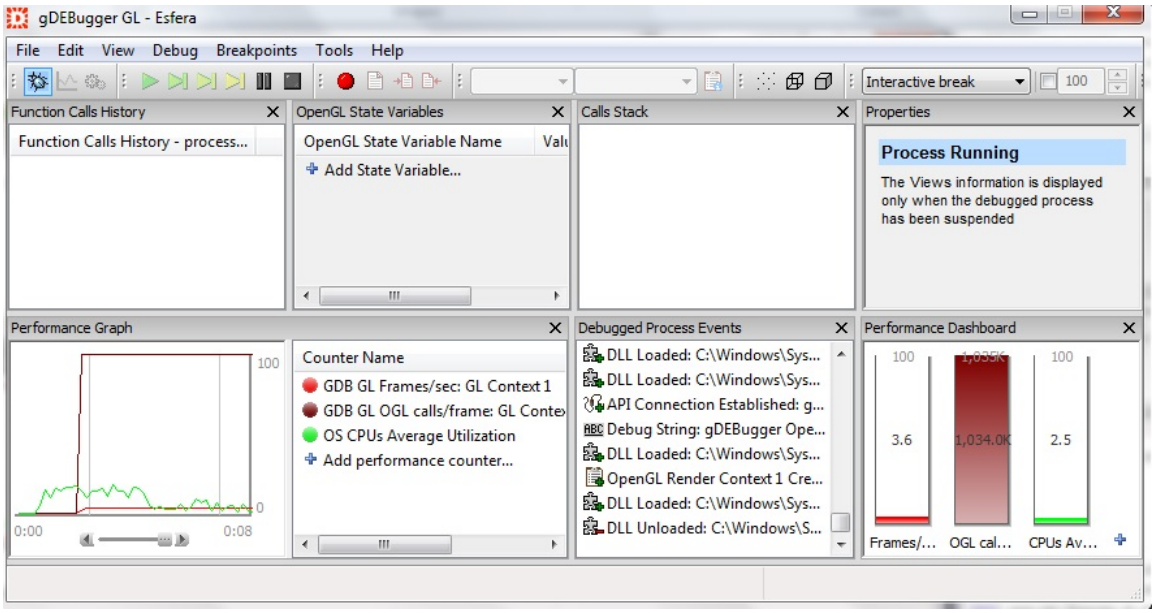

Figura 13 – Ferramenta *gDEBugger*: Gráfico de desempenho, histórico de chamadas, valor das variáveis e depuração

3.4 Ensaios Iniciais

Primeiramente foi analisado se era factível estender o tema também para a plataforma *Android* – tanto no que diz respeito ao prazo quanto em relação ao conhecimento prévio da autora deste trabalho. Então avaliou-se o o nível de dificuldade de implementação de um *shader* para plataforma *Android* (principalmente pelo fato da estudante 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, as quais foram apresentadas na subseção Configuração do Ambiente. 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.

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, 2013). 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 foi 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*, o que demandará mais tempo para implementá-la.

A seguir serão detalhados os *shaders* escolhidos, implementados e analisados nos testes de viabilidade.

3.4.1 *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 no código 3.1.

Listing 3.1 – *Shader* cor vermelha: *vertex shader*

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

uniform float time;

void main()
{
```

```
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;

}
```

Já o seu *fragment shader* (código 3.2) estabele que todo fragmento possui a cor vermelha, por meio da palavra restrita *gl_FragColor*.

Listing 3.2 – *Shader* cor vermelha: *fragment shader*

```
// Sets fragment color to red.

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

O resultado da aplicação deste *shader* é mostrado na Figura 14, em que a cor das esferas é vermelha e cada uma delas faz distintas movimentações.

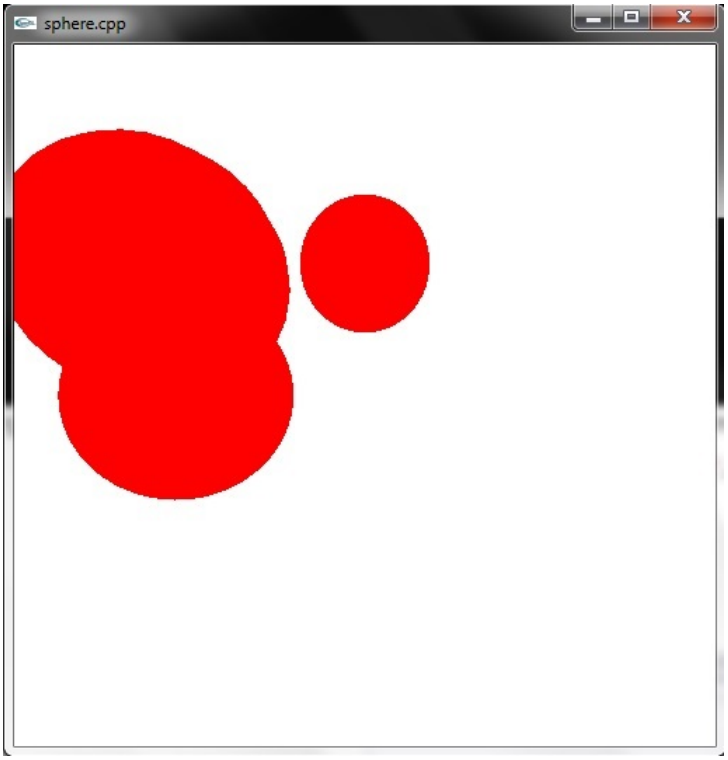

Figura 14 – *Red shader*

3.4.2 *Flatten shading*

A ideia do *flatten shading* é tornar o modelo tridimensional em bidimensional, achatado. Para isso, a coordenada z deve ser definida como zero. Mas para dar movimentação à malha do objeto, como mostrado no código 3.3, 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).

Listing 3.3 – *Flatten Shader*: *vertex shader*

```
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;

}
```

Neste caso o *fragment shader* não interfere no resultado desejado, de modo que foi utilizado o mesmo do *red shader*, definindo a cor para vermelha. A Figura 15 mostra as esferas achatadas, com a coordenada z variando de acordo com o fator definido.

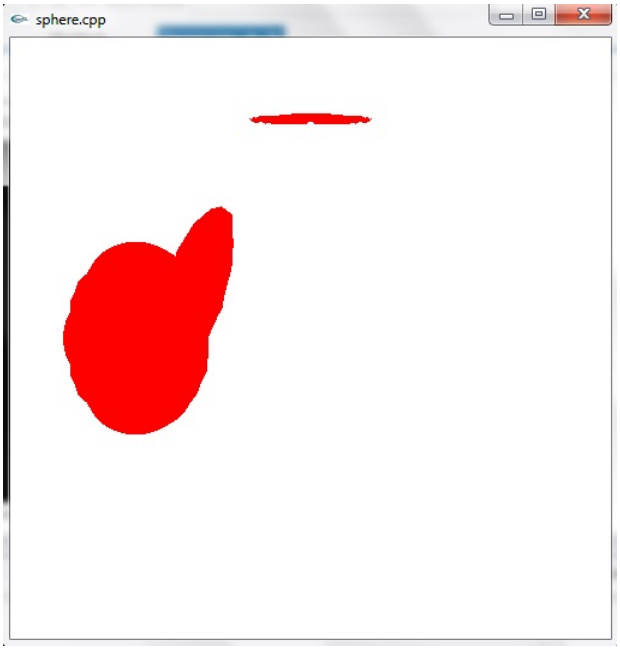

Figura 15 – *Flatten shader*

3.4.3 *Toon shading*

O *toon shading* calcula a intensidade da luz por vértice para escolher uma das quatro cores pré-definidas. No código 3.4 é mostrado 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 *gl_Normal*).

Listing 3.4 – *Toon Shader*: *vertex shader*

```
uniform vec3 lightDir;

varying float intensity;

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

A variável *intensity* do tipo *varying* é passada do *vertex shader* para o *fragment shader* para determinar qual das quatro cores será escolhida (código 3.5).

Listing 3.5 – *Toon Shader*: *fragment shader*

```
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;
        }
```

Assim, a direção da luz passada pelo programa é $(0, 1, 1)$ e o resultado da aplicação do *shader* é mostrado na Figura 16.

3.4.4 *Phong shading*

O *vertex* e *fragment shaders* do *phong shading* implementam a técnica descrita na Seção 2.7, em que primeiramente interpolam-se os valores das normais das primitivas e então computam-se os cálculos de luz para cada *pixel*, utilizando as normais interpoladas. Os códigos 3.6 e 3.7 mostram as definições do *vertex* e *fragment shaders*, respectivamente,

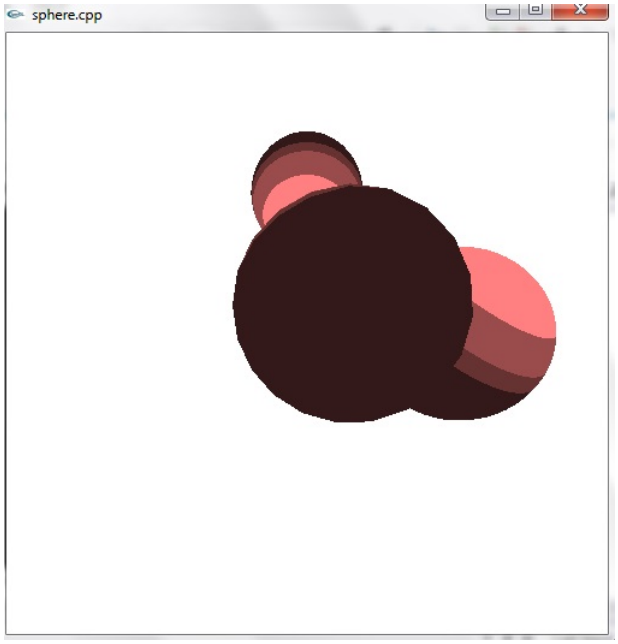

Figura 16 – *Toon shader*

em que para isso é necessário definir as propriedades do material pelo programa.

Listing 3.6 – *Phong Shader*: *vertex shader*

```
//Vertex Shader
varying vec4 eyePosition;
varying vec3 diffuseColor,specularColor,emissiveColor;
varying vec3 ambientColor,normal;
varying float shininess;

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;
}
```

Listing 3.7 – *Phong Shader*: *fragment shader*

```
//Fragment shader
varying vec4 eyePosition;
varying vec3 normal,diffuseColor,specularColor;
varying vec3 emissiveColor,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;
}
```

O resultado é mostrado na Figura 17, o qual é muito parecido com o resultado padrão implementado pela *OpenGL*.

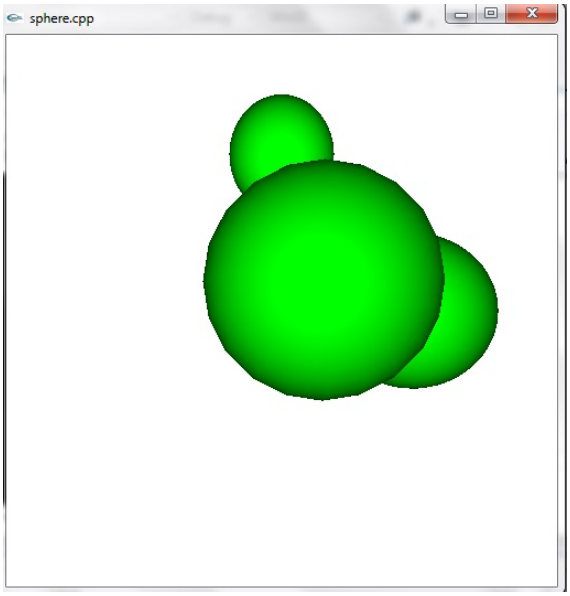

Figura 17 – *Phong shader*

3.4.5 *Texture shading*

O *vertex shader* do *texture shading* primeiramente armazena as coordenadas de textura numa variável do tipo *varying* (código 3.8), e as repassa 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 quádrico[1], que por sua vez dá opção de criar coordenadas de textura (utilizadas pelo *shader*).

Listing 3.8 – *Texture Shader*: *vertex shader*

```
varying vec2  TexCoord;

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

Figura 18 – Textura utilizada

No código 3.9, o *fragment shader* por sua vez, utiliza a textura passada pelo programa (Figura 18) e aplica na coordenada repassada pelo *vertex shader*.

Listing 3.9 – *Texture Shader*: *fragment shader*

```
uniform sampler2D myTexture;
varying vec2 TexCoord;

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

O resultado é mostrado na Figura 19.

[1] Um objeto do tipo quádrico representa uma superfície quádrica, em que as coordenadas formam um polinômio de segundo grau de no máximo três variáveis, como por exemplo, as superfícies cônicas, esféricas e cilíndricas.

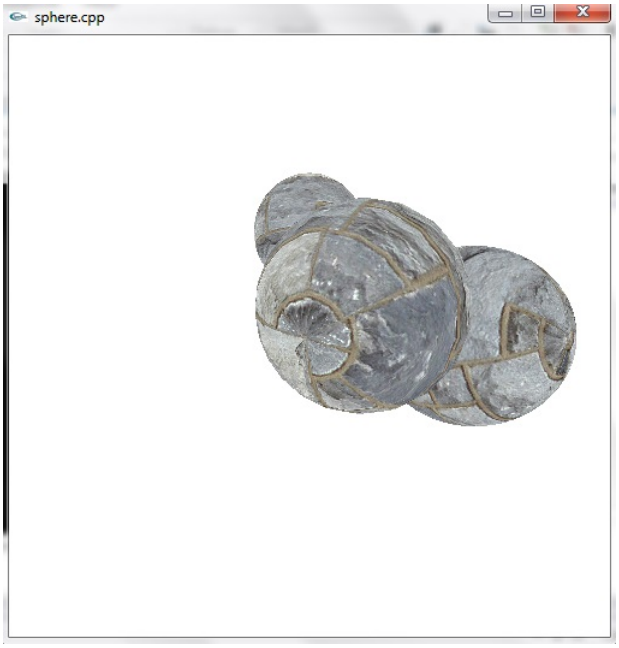

Figura 19 – *Texture shader*

Terminadas as implementações, foram utilizadas as ferramentas mencionadas anteriormente, sendoi possível coletar o número de quadros por segundo para diferentes números de polígonos. E dessa forma, gráficos (quadros por segundo *versus* número de polígonos) para cada *shader* implementado foram traçados, podendo então analisar experimentalmente suas complexidades algorítmicas.

3.5 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 2.4.3 e o *shader* mencionado. O resultado se encontra na Figura 20.

Este teste gerou a confiança de que seria possível implementar para a plataforma *Android* dentro do prazo estipulado.

3.6 Implementação Computador

A fim de se ter uma comparação com o computador, já que a complexidade algorítmica não deve mudar no que diz respeito à ordem, codificou-se o mesmo programa mostrado na Figura 21 , porém utilizando a biblioteca OpenGL, Glut e a linguagem de programação C++.

Figura 20 – Implementação plataforma *Android*

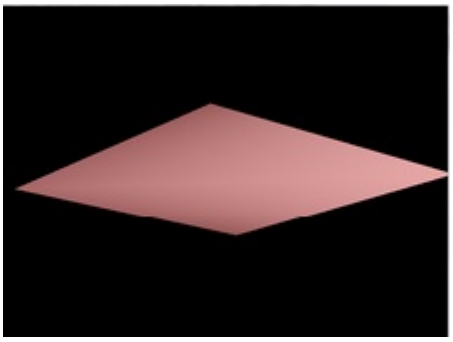

Figura 21 – Implementação Computador

3.7 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 do formato *obj*.

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

Após a implementação dos *shaders*, foi utilizada a ferramenta *gDEBugger* descrita na Seção 3.3 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 algorítmicas dos *shaders* através das curvas plotadas.

A Figura 22 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 CSV.

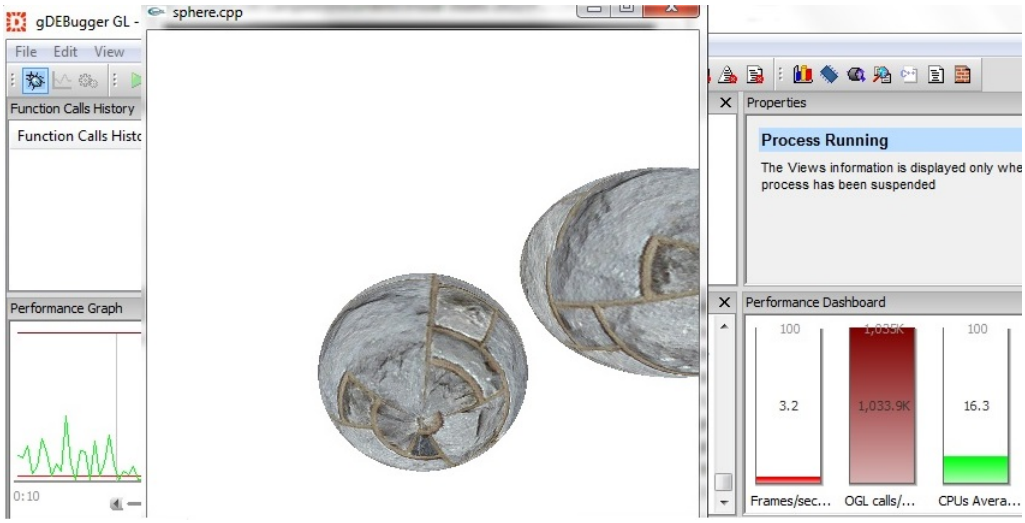

Figura 22 – Ferramenta *gDEBugger* sendo utilizada

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 Equação (4.1), em que y é o número de polígonos, s é o número de subdivisões e n 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).

$$\mathcal{Y} = s^2 \cdot n \cdot 2 \tag{4.1}$$

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*. A Tabela 7, Tabela 8 e Tabela 9 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**	***Red Shader***	***Flatten Shader***
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 7 – Quadros por segundo para o *Red Shader* e o *Flatten Shader*

Nº de subdivisões	**Nº de polígonos**	***Toon Shader***	***Phong Shader***
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 8 – Quadros por segundo para o *Toon Shader* e *Phong Shader*

Nº de subdivisões	**Nº de polígonos**	***Texture Shader***
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 9 – Quadros por segundo para o *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 (Figura 23), que visualmente surege uma curva 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 tem visualização mais difícil.

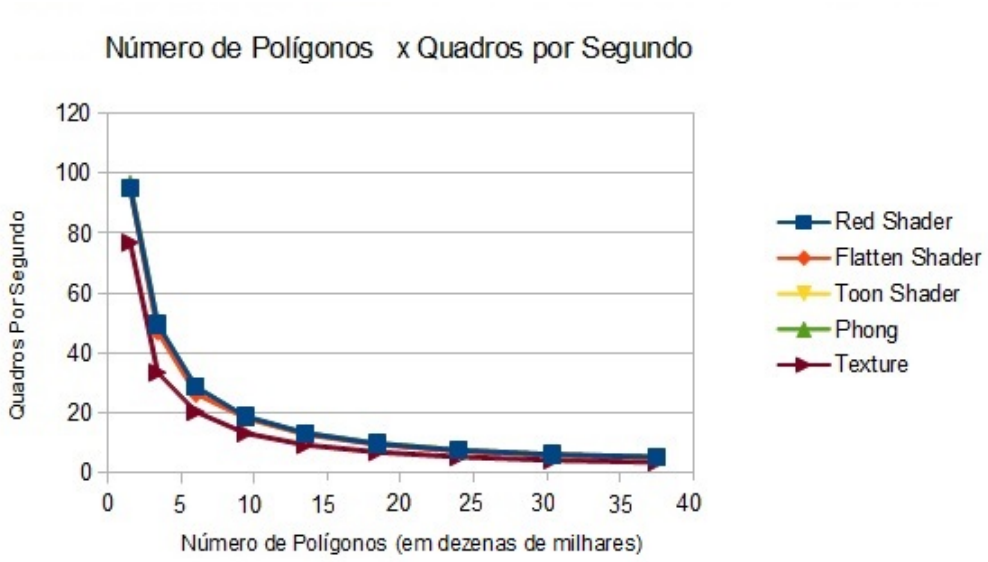

Figura 23 – Complexidade algoritma: exponencial

De acordo com a Seção 2.9, foi visto que quando aplica-se a função logarítmica nos dois lados da equação da exponencial, ela se torna uma reta. Assim, para confirmar experimentalmente se a curva obtida dos gráficos é realmente uma exponencial, traçaram-se os gráficos novamente na Figura 24, mas dessa vez, na escala logarítmica.

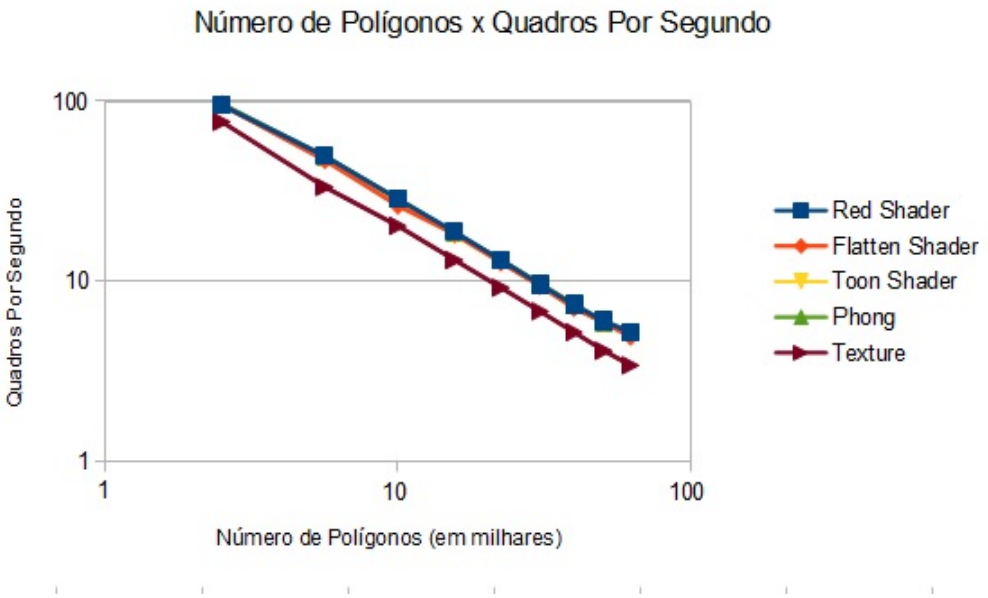

Figura 24 – *Complexidade Algorítmica: reta*

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 de ordem exponencial.

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 mostrou potencial, identificando como ordem de complexidade dos *shaders* uma curva exponencial. 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 2.9 – será utilizado para estimar os parâmetros da curva exponencial específicos de cada *shader*, para cada dispositivo, uma vez que, embora as curvas sejam todas de mesma família, serão justamente estes parâmetros que determinarão as particularidades de cada dispositivo/ambiente. Estas curvas permitirão a estimativa da 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*, o que ficou evidenciado no momento da conversão de um programa escrito para *mobile* para o 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*.

6 Cronograma de Desenvolvimento

O cronograma abaixo engloba as atividades já realizadas e também as que ainda serão feitas na continuidade do trabalho.

Atividades	**Início**	**Fim**
Configuração do Ambiente		
Configurar desenvolvimento para plataforma *Android*	19/08/2013	26/08/2013
Configurar desenvolvimento para computador	10/09/2013	16/09/2013
Levantar ferramentas de otimização gráfica	17/09/2013	23/09/2013
Configurar ferramentas de otimização gráfica	24/09/2013	27/09/2013
Definição do Tema		
Implementar *shader* para plataforma *Android*	27/08/2013	09/09/2013
Implementar *shader* para computador	17/09/2013	07/10/2013
Implementar *shaders* para análise de complexidade algorítmica	08/10/2013	21/10/2013
Plotar gráficos para análise algorítmica	22/10/2013	28/10/2013
Implementação dos *shaders*		
Modelar objetos tridimensionais	06/01/2014	25/02/2014
Implementar leitor de formato obj	20/01/2014	03/02/2014
Selecionar e estudar *shaders*	10/03/2014	20/05/2014
Implementar *shaders*	20/03/2014	25/05/2014
Plotar gráficos para análise algorítmica	26/05/2014	02/06/2014
Analisar e ajustas curvas obtidas	03/06/2014	13/06/2014
Confecção do trabalho escrito		
Escrever TCC 1	04/11/2014	24/11/2014
Escrever TCC 2	01/06/2014	09/07/2014

Tabela 10 – Atividades do cronograma de desenvolvimento

Referências

ANGEL, E.; SHREINER, D. *Interactive Computer Graphics*: A top-down approach with shader-based opengl. 6. ed. Boston, Massachusetts: Pearson, 2012. Citado na página 27.

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 15.

BROTHALER, K. *OpenGL ES 2 for Android*: A quick-start guide. 1. ed. Dallas, Texas: The Pragmatic Bookshelf, 2013. Citado na página 27.

DROZDEK, A. *Estrutura de Dados e Algoritmos em C++*. 2. ed. São Paulo, São Paulo: Cengage Learning, 2002. Citado na página 32.

GASTER, B. et al. *Heterogeneous Computing with OpenCL*. 1. ed. Waltham, Massachusetts: Morgan Kaufmann, 2013. Citado na página 37.

GREGORY, J. *Game Engine Architecture*. 2. ed. Boca Raton, Florida: CRC Press, 2009. Citado na página 28.

GUHA, S. *Computer Graphics Through OpenGL*: From theory to experience. 1. ed. Boca Raton, Florida: CRC Press, 2011. Citado 4 vezes nas páginas 21, 25, 26 e 31.

JACKSON, W. *Learn Android App Development*. 1. ed. New York, New York: Apress, 2013. Citado na página 19.

LEITHOLD, L. *O Cálculo com Geometria Analítica*. 3. ed. São Paulo, São Paulo: Harbra, 1994. Citado na página 33.

LINUX, M. P. *glutSolidSphere*. 2013. Disponível em: < http://www.pkill.info/linux/man/3-glutSolidSphere/ >. Acessado em: 30 out. 2013. Citado na página 38.

MOLLER, T. A.; HAINES, E.; HOFFMAN, N. *Real-Time Rendering*. 2. ed. Boca Raton, Florida: CRC Press, 2008. Citado 2 vezes nas páginas 21 e 29.

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 15.

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 35.

RORRES, A. *Álgebra Linear com Aplicações*. 8. ed. Porto Alegre, Rio Grande do Sul: Bookman, 2001. Citado na página 33.

SANDBERG, R.; ROLLINS, M. *The Business of Android Apps Development*. 2. ed. New York, New York: Apress, 2013. Citado 2 vezes nas páginas 15 e 19.

SEDGEWICK, R. *Algorithms in C*. 1. ed. Westford,Massachusetts: Addison-Wesley, 1990. Citado na página 32.

SHERROD, A. *Game Graphics Programming*. 1. ed. Boston, Massachusetts: Course Technology, 2011. Citado 2 vezes nas páginas 15 e 28.

SMITHWICK, M.; VERMA, M. *Pro OpenGL ES for Android*. 1. ed. New York, New York: Apress, 2012. Citado na página 21.

WRIGHT, R. S. et al. *OpenGL SuperBible*: Comprehensive tutorial and reference. 5. ed. Boston, Massachusetts: Pearson, 2008. Citado 2 vezes nas páginas 20 e 21.