

Computação Gráfica: Etapa#1

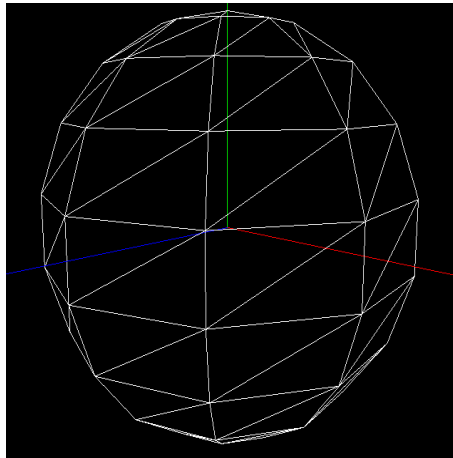
Bernardo Rodrigues
a79008@alunos.uminho.pt

César Silva
a77518@alunos.uminho.pt

Pedro Faria
a82725@alunos.uminho.pt

Rui Silva
a77219@alunos.uminho.pt

Universidade do Minho — 25 de Fevereiro de 2019



Resumo

A **Computação Gráfica**, uma das muitas áreas da informática, que assume um papel fulcral em interações homem-máquina e na visualização de dados. o seu espectro de aplicações varia desde geração de imagens até a simulação do mundo real.

O presente relatório refere-se à primeira fase de entrega da componente prática da Unidade Curricular de **Computação Gráfica** enquadrada no curso de *Ciências da Computação* da *Universidade do Minho*.

Conteúdo

| | | |
|----------|----------------------------------|-----------|
| 1 | Introdução | 4 |
| 1.1 | Estrutura do Relatório | 4 |
| 2 | Gerador | 5 |
| 2.1 | Caixa | 5 |
| 2.2 | Cone | 5 |
| 2.3 | Esfera | 7 |
| 2.4 | Plano | 8 |
| 3 | Motor | 9 |
| 4 | Conclusão | 11 |
| A | Código do Gerador | 12 |
| B | Código do Motor | 13 |

1 Introdução

Nesta fase, foram desenvolvidas duas aplicações. Um **Gerador** de modelos, que aceite argumentos a partir do terminal, com a função de gerar os pontos de uma primitiva gráfica desejada pelo utilizador, imprimindo estes para um ficheiro. E a última, um **Motor** que interpreta os pontos gerados anteriormente de acordo com um ficheiro de configuração dado. Com base nos tópicos enunciados e ferramentas exploradas nas aulas implementamos o **Gerador** e o **Motor** na linguagem **C++**. Em particular esta última faz uso de biblioteca **TinyXML2** para que a leitura dos documentos que lhe são dados com input seja feita de forma simples e consistente. E por fim, utilizamos a API fornecida pelo **OpenGL** para dar vida aos nossos modelos.

1.1 Estrutura do Relatório

O presente documento divide-se em duas partes principais. A primeira onde expomos brevemente os conteúdos do trabalho, explicamos funcionalidades, fazemos considerações importantes e demonstração de resultados. No fim, em anexo dispomos todo o código interveniente dos diferentes módulos do projeto.

2 Gerador

O nosso **Gerador** é um pequeno programa em C++ , cuja implementação é disponibilizada em anexo, que depois de compilado, o correspondente executável escreve para um ficheiro (um por linha) os vértices da primitiva gráfica desejada como iremos ilustrar nas seguintes secções.

2.1 Caixa

Para geração desta primitiva o utilizador deverá invocar a aplicação a com o nome do executável seguido dos comprimentos dos lados de um paralelepípedo no eixos com X's, Y's e Z's respetivamente e por fim o nome do ficheiro destino. A sintaxe é demonstrada no exemplo abaixo:

Command Line

```
$ g++ -o gerador gen.cpp
$ ./gerador caixa 3 4 5 osmeusvertices
$ ls
$ gen.cpp          gerador          osmeusvertices.txt
```

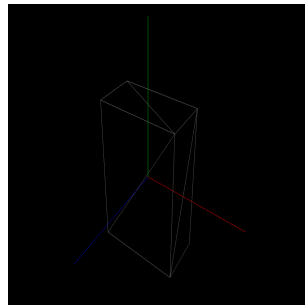


Figura 1: Caixa



Notice: Todas as faces são desenhadas com dois triângulos apontados para o exterior pela norma da mão direita mencionada nas aulas. Um zoom excessivo, ou a escolha de dimensões superiores à distância da câmara à origem(onde este é centrado) pode levar ao aparente desaparecimento do modelo.

2.2 Cone

O **Cone** recebe como argumentos o raio da base, a sua altura, o número de slices e stacks.

A construção deste começa por fixar um uma *slice* calculando os vértices da base correspondente, de seguida todas as *stacks* relativas, sendo a última stack - a do bico - um caso especial.

O algoritmo usa noções como semelhança de triângulos para cálculo dos sucessivos raios das circunferências formadas pelas *stacks*.



Info: Apresentamos o significado das variáveis usadas no programa que gera os pontos do **Cone**:

stk - diferença entre stacks consecutivas

slcd - diferença entre slices consecutivas

raiod - diferença entre o raio de duas stacks consecutivas

stk - stack atual

slc - slice atual

nslc - próxima slice

nstk - próxima stack

nr - próximo raio

r - raio atual

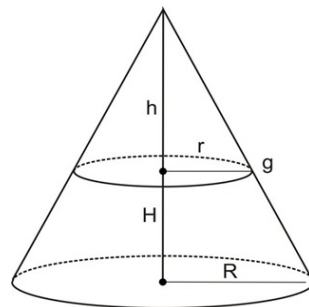
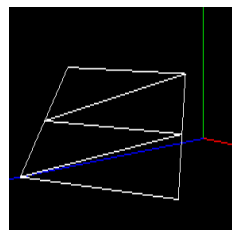
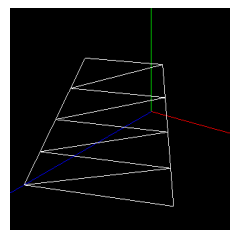


Figura 2: Semelhança de triângulos num cone

Apresentamos algumas imagens que ilustram a criação do cone.

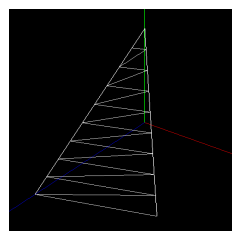


(a) 2 stacks

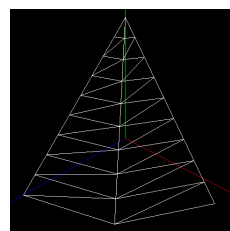


(b) 4 stacks

Figura 3: Progressão das stacks do cone



(a) 1 slice



(b) 2 slices

Figura 4: Progressão das slices do cone

Finalmente obtemos:

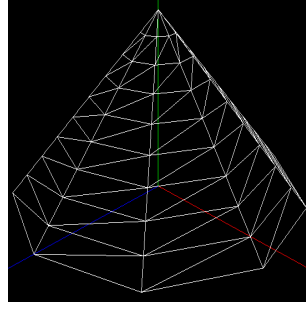


Figura 5: Cone

2.3 Esfera

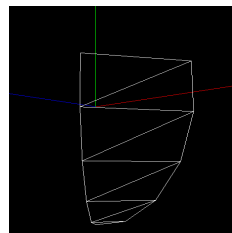
A **Esfera** recebe como argumentos o seu raio, o número de slices e stacks. A sua construção usa coordenadas esféricas usando o raio dado como argumentos e manipulando 2 ângulos *Alfa* e *Beta*. *Alfa* é dado por:

$$alfa = \frac{2 \times \Pi}{slices}$$

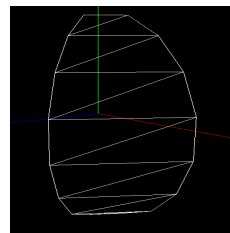
Este nos programas é usado juntamente com o raio para calcular os pontos de *slices* consecutivas. De seguida *beta* é dado por:

$$beta = \frac{\Pi \div 2}{stacks}$$

Este desempenha uma função igual ao anterior considerando *stacks*.

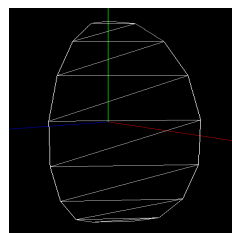


(a) 4 stacks

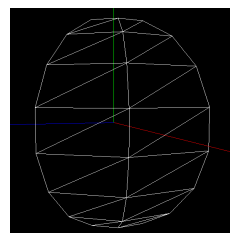


(b) 8 stacks

Figura 6: Progressão das stacks da esfera



(a) 1 slice



(b) 2 slices

Figura 7: Progressão das slices da esfera

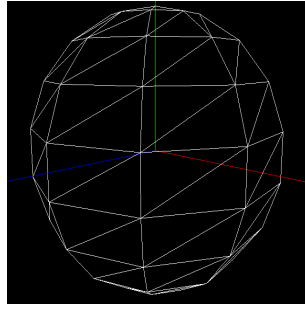


Figura 8: Esfera

2.4 Plano

O requisito estabelecido no guião do trabalho veio em muito simplificar a representação do plano XZ ao ponto de para a sua computação seja apenas necessário um único argumento que representa o tamanho da porção visível desejada.

Command Line

```
$ g++ -o gerador gen.cpp
$ ./gerador plano 5 pontos
$ ls
$ gen.cpp          gerador          pontos.txt
```

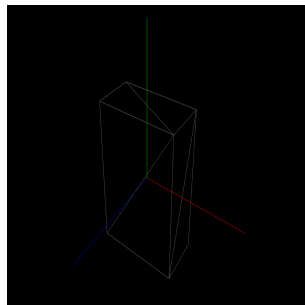


Figura 9: Caixa



Notice: Os planos são infinitos a referência ao tamanho é feita apenas para facilitar a observação do mesmo. São desenhados 4 triângulos, em vez de dois, para que esta primitiva seja visível de todas as perspectivas.

3 Motor

O motor é a parte do nosso trabalho que faz o linking de todas as outras partes que foram realizadas. O motor, lê um ficheiro xml (conf.xml). Este ficheiro contém uma estrutura bastante simples, dividida em scenes.

```
conf.xml

<scene>
    <model file="esfera.txt" />
    <model file="cone.txt" />
    <model file="caixa.txt" />
</scene>
```

Cada ficheiro que está referenciado nas tags **<model>** é um ficheiro de texto criado pelo nosso gerador e contém todos os pontos das figuras que queremos representar. Como o motor lê os ficheiros de pontos a partir dum ficheiro **XML** utilizamos um parser xml para C++ chamado **tinyxml-2**.

```
main.cpp

...
tinyxml2::XMLDocument doc;

doc.LoadFile("./conf.xml");

tinyxml2::XMLNode *scene = doc.FirstChild();

tinyxml2::XMLElement* model;

while(scene) {
    for(model = scene->FirstChildElement();
        model != NULL;
        model = model->NextSiblingElement()) {
        const char * file;
        file = model->Attribute("file");
        guardaPontos(file);
    }
    scene = scene->NextSiblingElement();
}
...
```

Com este snippet de código, criamos um objeto do tipo XMLDocument. De seguida, com a função **LoadFile**, abrimos o ficheiro de configuração **conf.xml**, e começamos a manipular o seu conteúdo. Criamos um objeto do tipo **XMLNode** e associamos-lhe a primeira **tag** do ficheiro conf.xml que é a raiz da estrutura do nosso ficheiro. No ciclo while, percorremos todos o ChildElements de scene, que são as tags que guardam os nossos ficheiros das figuras. Cada ficheiro de figura tirado dos **models** é passado à função **guardaPontos**, que será explicada de seguida.

```

main.cpp

...
struct Pontos {
    float a;
    float b;
    float c;
};

std::vector<Pontos> pontos;

void guardaPontos(std::string ficheiro) {
    std::ifstream file;
    std::string s = "./";
    s.append(ficheiro.c_str());
    file.open(s.c_str());
    float a,b,c;
    while(file >> a >> b >> c) {
        Pontos aux;
        aux.a = a;
        aux.b = b;
        aux.c = c;
        pontos.push_back(aux);
    }
}
...

```

Criamos uma estrutura **Pontos** que tem como campos 3 *floats*, que servem para registar as coordenadas destes. De seguida usamos um vector que utiliza a estrutura enunciada para os guardar. À função **guardaPontos** são passados nomes de ficheiros. A função abre os ficheiros e lê pontos linha a linha, guardando cada coordenada *x*, *y* e *z* num **Ponto** e de seguida inserindo-o no vector.



Info: A instrução:

```
file >> a >> b >> c
```

associa a cada uma das variáveis *a*, *b* e *c*, as coordenadas da linha que está a ser lida em cada iteração do ciclo.

Por ultimo temos a função **printPontos**:

```

main.cpp

...
void printPontos(std::vector<Pontos> pontos) {
    for(int i = 0; i < pontos.size(); i++) {
        glVertex3f(pontos[i].a,
                  pontos[i].b,
                  pontos[i].c);
    }
}
...

```

Esta função é chamada na `renderScene` e gera todos os pontos percorrendo o vector.

4 Conclusão

De um modo geral achamos que grupo, apesar das dificuldades, correspondeu às exigências propostas do enunciado. Tentamos implementar uma solução que suportasse requisitos futuros.

A Código do Gerador

B Código do Motor