Computação Gráfica (3º ano de Curso)

Fase 1

Relatório de Desenvolvimento

Diogo Fernandes (A87968)
Luís Guimarães (A87947)
Ivo Lima (A90214)

14 de março de 2021

Conteúdo

Capítulo 1

Introdução

Esta fase consistiu no desenvolvimento de dois programas. O primeiro, *generator*, é responsável pela geração de ficheiros com as informações relativas aos modelos que se pretendem criar, e o segundo, *engine*, por ler essas informações para memória segundo um ficheiro de configuração.

A linguagem de programação utilizada para escrever cada um dos programas foi C++, e os ficheiros de configuração devem estar escritos em xml.

Capítulo 2

Generator

Este programa trata de gerar ficheiros com as informações relativas aos modelos que se pretendem criar.

2.1 main

A main do *generator* é responsável por reconhecer qual figura que se pretende gerar e encaminhar o programa para a função correspondente. Trata também de passar os parâmetros para as funções geradoras, assim como controlar possíveis erros relacionados com os argumentos do programa.

As diferentes mensagens de erro que podem ser geradas são:

- "Error! No arguments passed.";
- "Error! Invalid number of arguments.";
- "Error! That figure doesn't exist.";

A main começa com três *strings*: *fname*, *fig*, *res* e uma *stringstream* *ss*. Os argumentos recebidos pela main serão passados a *ss* separados por espaços. Dependendo da ordem pela qual os argumentos foram escritos, na ausência de anomalias, *ss* atribuirá a *fig* o argumento que identifica a figura e ao *fname* o nome que o ficheiro criado deverá ter. Dependendo da figura a ser gerada, serão criadas mais variáveis que alojarão os valores a serem passados como parâmetros à função geradora.

Cada uma das funções geradoras, escreve o número de triângulos e todos os pontos gerados numa única *string* que é posteriormente retornada à main que invoca a função *constructF*, responsável por construir o ficheiro resultado.

Código da main:

```
int main(int argc, char* argv[]){
    string fname, fig, res;
    stringstream ss;
```

```
for (int i = 1; i< argc; i++){
    ss << argv[i];
    ss << " ";
}
ss >> fig;
if (argc == 1){
  cout << "\nError! No arguments passed.\n";
  exit(1);
}
switch (fig.at(0)) {
    case 'p': {
        if(argc < 3){
            cout << "\nError! Invalid number of arguments.\n
";
            exit(1);
        }
        ss >> fname;
        res = gen_Plane(PLANE_HEIGHT);
    break;
    }
    case 'b': {
        if(argc < 6){
            cout << "\nError! Invalid number of arguments.\n
";
            exit(1);
        }
        float pBx; ss >> pBx;
        float pBy; ss >> pBy;
        float pBz; ss >> pBz;
        ss >> fname;
        res = gen_Box(pBx,pBy,pBz);
    break;
    }
    case 's': {
        if(argc < 6){
          cout << "\nError! Invalid number of arguments.\n";
          exit(1);
        }
        float radius; ss >> radius;
        int slices; ss >> slices;
        int stacks; ss >> stacks;
        ss >> fname;
        res = gen_Sphere(radius,slices,stacks);
    break;
    }
    case 'c': {
        if(argc < 7){
          cout << "\nError! Invalid number of arguments.\n";
          exit(1);
        }
        float radius; ss >> radius;
        float height; ss >> height;
        int slices; ss >> slices;
        int stacks; ss >> stacks;
```

```
            ss >> fname;
            res = gen_Cone(radius,height,slices,stacks);
        break;
        }
        default:{
                cout << "\nError! That figure doesn't exist.\n
    ";
        break;
        }
    }
        constructF(res,fname);
        return 0;
}
```

2.2 Plane

A função responsável pela geração do plano é *gen_Plane*.

Esta função gera, com uma dimensão constante, 4 triângulos de forma a que o plano consiga ser visto dos dois lados com os seguintes pontos:

```
(width , 0, width);
(width , 0, -width);
(-width, 0, -width);

(width, 0, width);
(-width, 0, -width);
(-width, 0, width);

(width, 0, -width);
(width, 0, width);
(-width, 0, -width);

(-width, 0, -width);
(width, 0, width);
(-width, 0, width);
```

2.3 Box

A função responsável pela geração da caixa é *gen_Box*. Esta função recebe como parâmetros 3 dimensões: *x, y e z*. Uma vez que a figura fica centrada na origem, cada dimensão é dividida por duas unidades de forma a que a dimensão total em x seja igual ao x passado como argumento.

Esta função gera 12 triângulos com os seguintes pontos:

```
//Top
(x, y, z);
(x, y, -z);
(-x, y, -z);

(x, y, z);
(-x, y, -z);
(-x, y, z);

//Bottom
(x, -y, -z);
(x, -y, z);
(-x, -y, -z);

(-x, -y, -z);
(x, -y, z);
(-x, -y, z);

//Right
(x, -y, -z);
(x, y, -z);
(x, y, z);

(x, -y, -z);
(x, y, z);
(x, -y, z);
```

```
//Left
(-x, y, -z);
(-x, -y, -z);
(-x, y, z);

(-x, y, z);
(-x, -y, -z);
(-x, -y, z);

//Front
(x, -y, z);
(x, y, z);
(-x, y, z);

(x, -y, z);
(-x, y, z);
(-x, -y, z);

//Rear
(x, y, -z);
(x, -y, -z);
(-x, y, -z);

(-x, y, -z);
(x, -y, -z);
(-x, -y, -z);
```

2.4 Sphere

A função que gera a esfera tem como nome *gen_Sphere* e recebe 3 parâmetros: *radius, slices e stacks.*

O cálculo dos pontos é feito com recurso a duas variáveis a e b que representam dois ângulos, em radianos. O ângulo a varia entre 0 e 2π em torno do eixo do y e o número de valores que toma depende proporcionalmente do parâmetro *slices*. O ângulo b toma valores entre -$\pi/2$ e $\pi/2$ e o número de valores que toma depende proporcionalmente do parâmetro *stacks*.

Para além destas variáveis, a função usa também *a_interval* e *b_interval* que, depois de calculados de acordo com os parâmetros, contêm as distâncias entre dois valores consecutivos de a e b, respetivamente. Estes valores são calculados da seguinte forma:

```
float a_interval = 2 * PI / slices;
float b_interval = PI / stacks;
```

Outras variáveis utilizadas são *next_a*, *next_b* e *triangles*, que representam, respetivamente, os próximos valores de a e de b e o número de triangulos criados.

Os pontos da esfera são calculados recorrendo a dois ciclos. Um interno que percorre os b, e um externo que percorre os a. Em cada iteração do ciclo interno, são atualizados os valores de a, de b, de $next\_a$, de $next\_b$, somadas duas unidades a *triangles* e calculados os seguintes pontos:

```
(radius * cos(next_b) * sin(next_a), radius * sin(next_b),
    radius * cos(next_b) * cos(next_a));
(radius * cos(next_b) * sin(a), radius * sin(next_b), radius *
    cos(next_b) * cos(a));
(radius * cos(b) * sin(next_a), radius * sin(b), radius * cos(b)
     * cos(next_a));

(radius * cos(b) * sin(next_a), radius * sin(b), radius * cos(b)
     * cos(next_a));
(radius * cos(next_b) * sin(a), radius * sin(next_b), radius *
    cos(next_b) * cos(a));
(radius * cos(b) * sin(a), radius * sin(b), radius * cos(b) *
    cos(a));
```

Por razões de estética, adicionamos estas linhas de código, que permitem remediar erros causados por floats:

```
if (next_a > 2 * PI) {
  next_a = 2 * PI;
}
if (next_b > PI / 2) {
  next_b = PI / 2;
}
```

2.5 Cone

A função que gera o cone tem como nome *gen_Cone* e recebe 4 parâmetros: *radius, height, slices e stacks*. Esta função gera um cone com a base centrada na origem.

Recorre às variáveis a que representa um ângulo entre 0 e 2π, h que representa uma altura e varia entre 0 e o parâmetro *height* e $next\_a$ e $next\_h$ que representam, respetivamente, os próximos valores de a e de h. Recorre também a *interval* que representa a distância entre dois a consecutivos e a *stack_height* que representa a distância entre dois h consecutivos. Estes valores são calculados da seguinte forma:

```
float interval = 2 * PI / slices;
float stack_height = height / stacks;
```

Através de um ciclo externo, percorre os valores de a, para construir a base, e através de um ciclo interno, percorre os valores de h, para construir

as paredes. A base é construída com um número de triângulos correspondente ao parâmetro *slices* e as paredes são construídas com um número de triângulos correspondente a 2 * *slices* * *stacks*. Em cada iteração do ciclo externo, são atualizados os valores de a e de $next\_a$ e calculados os seguintes pontos:

```
(0.0f, 0, 0.0f);
(radius * sin(next_a), 0, radius * cos(next_a));
(radius * sin(a), 0, radius * cos(a));
```

E em cada iteração do ciclo interno, são atualizados os valores de h e de $next\_h$ e calculados os seguintes pontos:

```
(radius * sin(next_a) * ((height - next_h) / height), next_h,
    radius * cos(next_a) * ((height - next_h) / height));
(radius * sin(a) * ((height - next_h) / height), next_h, radius
    * cos(a) * ((height - next_h) / height));
(radius * sin(next_a) * ((height - h) / height), h, radius * cos
    (next_a) * ((height - h) / height));

(radius * sin(a) * ((height - next_h) / height), next_h, radius
    * cos(a) * ((height - next_h) / height));
(radius * sin(a) * ((height - h) / height), h, radius * cos(a) *
     ((height - h) / height));
(radius * sin(next_a) * ((height - h) / height), h, radius * cos
    (next_a) * ((height - h) / height));
```

Assim como no caso da esfera, por razões de estética, de forma a remediar erros causados por operações que envolvem floats, foram adicionadas as seguintes linhas de código no início dos ciclos correspondentes:

```
if (next_a > 2 * PI) {
  next_a = 2 * PI;
}
(...)
if (next_h > height) {
  next_h = height;
}
```

Capítulo 3

Engine

Este programa lê uma configuração previamente escrita em xml. Esta configuração deverá indicar o nome dos ficheiros previamente gerados com recurso ao *generator*.

O ficheiro indicado é carregado para a memória e escrito num vector<float>v. De seguida, é criado um Vertex Buffer Object (VBO) com o conteúdo de v e o número de vértices a gerar é calculado dividindo o valor escrito na primeira linha do ficheiro de leitura por 3.

Recorrendo a funções do openGL e às extensões glut e glew, os triângulos são desenhados a partir do VBO criado, utilizando assim a placa gráfica, e em modo *wireframe*.

Foram também incluídos no código do *engine* os ficheiros "tinyxml2.h"e "tinyxml2.cpp"que auxiliam no *parsing* do ficheiro de configuração.

Capítulo 4

Conclusão

A nível geral, e tendo em conta o que foi explicado nos capítulos anteriores, podemos afirmar que temos um projeto que numa fase inicial está bem conseguido. Acreditamos que respondemos de forma correta aos problemas que nos foram impostos tal como a de geração dos ficheiros .3d e a demostração das figuras numa janela.

Mas o tempo foi um dos nossos maiores inimigos fazendo com que certas questões relacionadas com o código e com o relatório tenham ficado um pouco menos polidas.

Todo este trabalho permitiu enriquecer e aprofundar aquilo que foi dado durante as aulas e ainda exigiu que despertássemos aptidões novas, pois esta foi a primeira vez que lidamos com ficheiros xml e que organizamos um cmake file para a criação da *engine*.