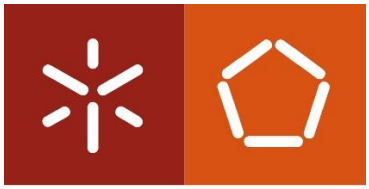

Universidade do Minho

Departamento de Informática

Computação Gráfica

Fase 1 - Grupo 7

José Ferreira (A83683)

João Teixeira (A85504)

Miguel Solino (A86435)

28 de Maio de 2020

Conteúdo

Capítulo 1

Introdução

Este trabalho foi desenvolvido no âmbito da unidade curricular de computação gráfica e está dividido em várias fases de entrega distintas sendo que esta é a primeira.
Esta fase está dividido em duas partes distintas, o *generator* e o *engine*.
O *generator* destina-se a gerar os pontos que desenham um conjunto de primitivas.
O *engine* destina-se a representar as primitivas produzidas anteriormente.
De forma a facilitar a compilação do código foi criado um *script* chamado *build.sh* que compila o código e gera os dois executáveis dentro da pasta *target/release/bin* quando executado.
Ao longo deste relatório irá ser explicado a metodologia e raciocínio usados para a realização desta fase.

Capítulo 2

Utils

2.1 Sistemas de Coordenadas

De forma a facilitar o desenho das formas foram desenvolvidas um conjunto de classes para representar dois sistemas distintos de coordenadas.
O primeiro sistema de coordenadas é o sistema cartesiano. Neste a posição de um ponto no espaço fica definido com 3 valores: x, y e z.

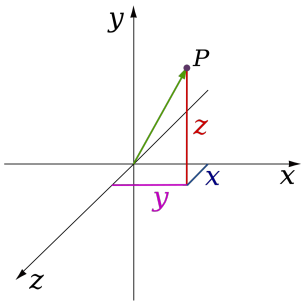

Figura 2.1: Coordenadas cartesianas

O segundo sistema de coordenadas é o sistema esférico. Neste, para definir a posição de um ponto no espaço, são também necessários 3 valores: raio, inclinação e azimute. O raio é a distancia que o ponto tem à origem do referencial, a inclinação é o ângulo que o ponto faz com o eixo vertical e o azimute é o ângulo que o ponto faz com o eixo dos z.

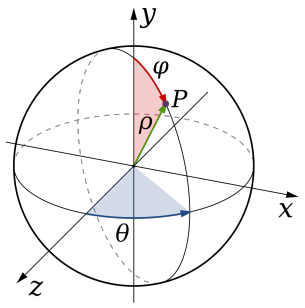

Figura 2.2: Coordenadas esféricas

Para ambos os sistemas de coordenadas foram definidos uma classe de pontos e uma classe de vetores. Também foram implementadas operações simples fazendo uso de *operator overloading* de forma a ser possível, por exemplo, escrever *Point* + *Vector* e o resultado desta operação dar um *Point*.

Capítulo 3

Generator

Neste módulo foram definidas 5 primitivas:

- Plano
- Caixa
- Esfera
- Cone
- Cilindro

O formato escolhido para representar os pontos gerados é o de colocar um ponto por linha com as coordenadas pela ordem X, Y e Z e separados por espaço.
Assim, por exemplo, o conjunto de pontos (0, 0, 0), (1, 0, 0) e (1, 1, 1) ficaria representado como.

```
0 0 0
1 0 0
1 1 1
```

3.1 Plano

Para desenhar um plano com dois triângulos sabendo o lado do plano basta conseguir obter as coordenadas dos cantos do plano.

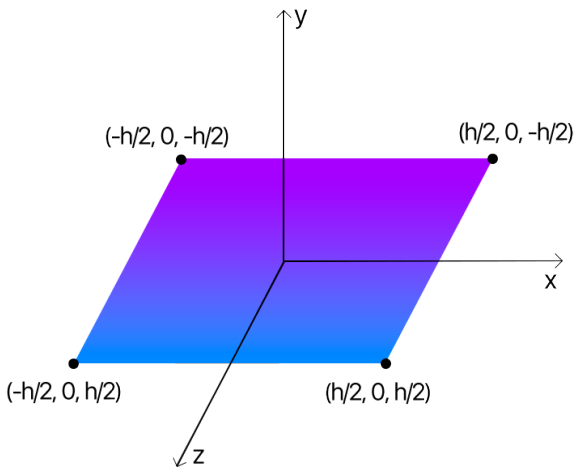

Figura 3.1: Esquema dos pontos no plano

Para desenhar um triângulo com o lado visível virado para cima é necessário seguir a regra da mão direita quando se decide porque ordem desenhar os pontos. Assim, os dois triângulos necessários são definidos da seguinte forma:
Para definir o triângulo superior:

$$(-h/2, \quad 0, \quad -h/2)$$
$$(-h/2, \quad 0, \quad h/2)$$
$$(h/2, \quad 0, \quad -h/2)$$

Para definir o triângulo inferior:

$$(h/2, \quad 0, \quad -h/2)$$
$$(-h/2, \quad 0, \quad h/2)$$
$$(h/2, \quad 0, \quad h/2)$$

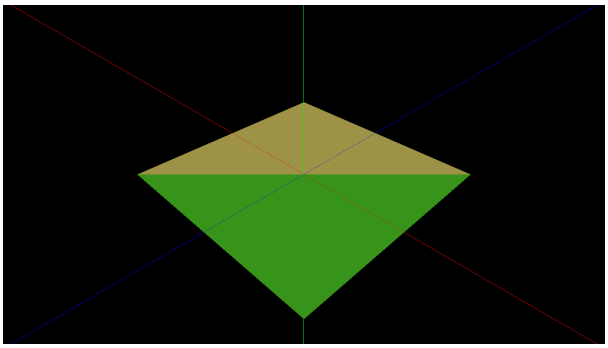

Figura 3.2: Plano Side:10

3.2 Caixa

Para desenhar uma caixa é necessário saber as dimensões da caixa ($\_X$, $\_Y$, $\_Z$) e, opcionalmente, o número de divisões da caixa (*_slices - 1*).
Para definir cada uma das faces de uma caixa é necessário ter um ponto que esteja nessa face. De forma a minimizar os pontos que são definidos escolhemos dois pontos em cantos opostos. Neste caso no canto inferior esquerdo da frente (*front*) e no canto superior direito de trás (*back*):

```
1 Point front  = Point(-_x/2, -_y/2,  _z/2);
2 Point back   = Point( _x/2,  _y/2, -_z/2);
```

Assim, todas as faces da caixa podem ser definidas com base em apenas dois pontos.

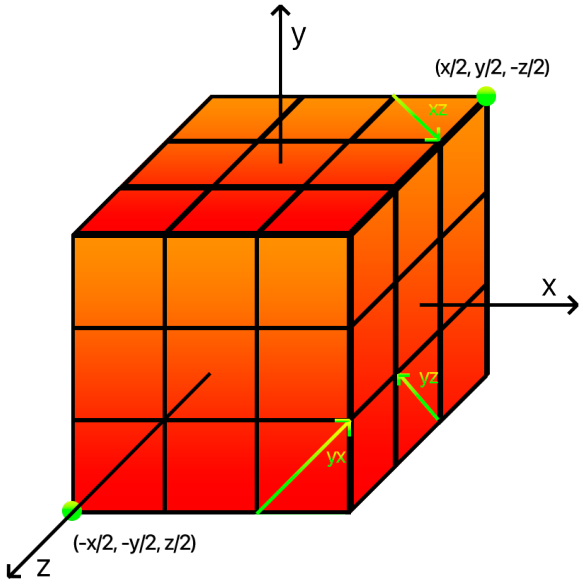

Figura 3.3: Vetores e pontos na caixa

Para facilitar as contas, o gerador converte as divisões que recebe no número de slices que o cubo tem (*slices = divisions + 1*).
Em seguida são definidos 3 vetores cada um associado a um dos planos que atravessam os eixos e têm como coordenadas o comprimento de cada uma das slices do cubo.

```
float stepx = _x / _slices;
float stepy = _y / _slices;
float stepz = _z / _slices;

Vector xy = Vector(stepx, stepy,     0);
Vector yz = Vector(0    , stepy, stepz);
Vector xz = Vector(stepx,     0, stepz);
```

Para obter os pontos ao longo de cada face utilizamos o produto de *haramard* sobre os vetores definidos anteriormente. Esta operação é definida para duas matrizes com as mesmas dimensões e produz uma matriz onde cada elemento (i, j) é o produto de elementos (i , j) das duas matrizes originais. isto permite escalar os vetores apenas num eixo.
Assim, o código para desenhar a face da frente da caixa fica da seguinte forma, sendo que o código para as restantes faces segue a mesma lógica:

```
for(i32 i = 0; i < _slices; i++) {
    for(i32 j = 0; j < _slices; j++) {
        //front
        coords.push_back(front + xy.hadamard(i  , j  , 0));
        coords.push_back(front + xy.hadamard(i+1, j  , 0));
        coords.push_back(front + xy.hadamard(i+1, j+1, 0));
        coords.push_back(front + xy.hadamard(i  , j  , 0));
        coords.push_back(front + xy.hadamard(i+1, j+1, 0));
        coords.push_back(front + xy.hadamard(i  , j+1, 0));
    }
}
```

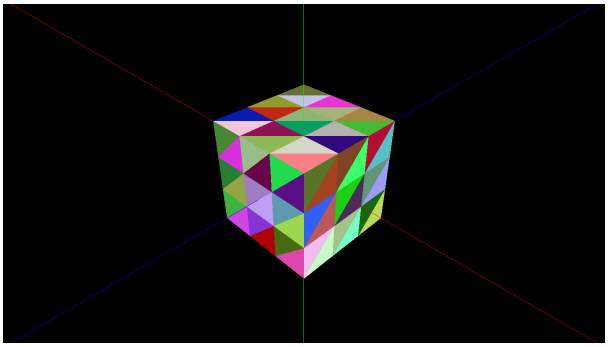

Figura 3.4: Caixa: X:5 Y:5 Z:5 Divisions:2

3.3 Esfera

Para desenhar a esfera é necessário saber o seu raio (*_radius*), o número de slices (*_slices*) e o número de stacks (*_stacks*).
Fizemos uso do sistema de coordenadas esféricas definidas anteriormente para desenhar a esfera. Como o próprio nome indica, o espaço definido por estas coordenadas é esférico o que facilita o desenho de uma esfera.
Tal como foi referido, um ponto precisa de três valores para ser defenido num sistema de coordenadas esféricas. O primeiro é a distância ao ponto do centro do referencial que por definição também é o raio da esfera. Assim basta obter a inclinação e o azimute.
Para facilitar estas contas são calculados dois valores:
O angulo entre cada slice (ang_slice = 2 * π / n de slices) e o angulo entre cada stack (ang_stack = π / n de stacks).

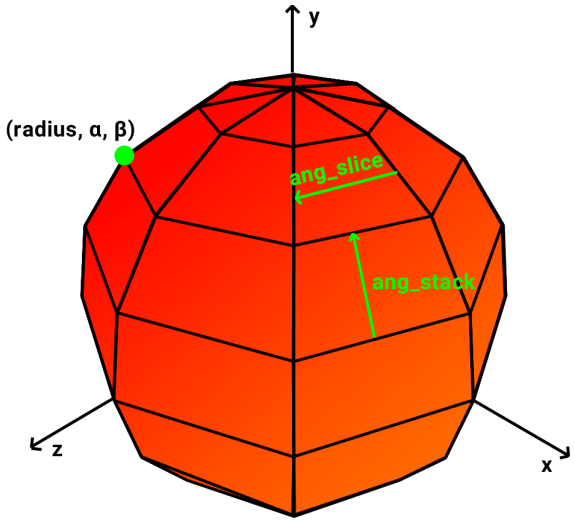

Figura 3.5: Angulos e pontos na esfera

Assim, para obter todos os pontos numa esfera podemos usar o excerto de código:

```
for(i32 slice = 0; slice < _slices; slice++)
    for(i32 stack = 0; stack < _stacks; stack++)
        coords.push_back(
            PointSpherical(_radius, ang_stack * stack, ang_slice * slice));
```

Para obter os próximos pontos da esfera de forma a criar os triângulos basta fazer *ang_stack * (stack + 1)* e *ang_slice * (slice + 1)*
Assim, juntando estes quatro pontos com o explicado na criação do plano:

```
for(i32 slice = 0; slice < _slices; slice++)
    for(i32 stack = 0; stack < _stacks; stack++) {
        //1st triangle
        coords.push_back(
            PointSpherical(_radius, ang_stack * (stack+1), ang_slice * (slice+1)));
        coords.push_back(
            PointSpherical(_radius, ang_stack * stack, ang_slice * (slice+1)));
        coords.push_back(
            PointSpherical(_radius, ang_stack * stack, ang_slice * slice));
        //2nd triangle
        coords.push_back(
            PointSpherical(_radius, ang_stack * (stack+1), ang_slice * slice));
        coords.push_back(
            PointSpherical(_radius, ang_stack * (stack+1), ang_slice
        * (slice+1)));
        coords.push_back(
            PointSpherical(_radius, ang_stack * stack, ang_slice
        * slice));
        }
```

O resultado deste código já gera uma esfera. No entanto, quando es está a desenhar a primeira e a última stack os triângulos são duplicados. Para resolver este problema basta não desenhar um dos triângulos nestes casos. Assim a versão final fica:

```
for(i32 slice = 0; slice < _slices; slice++)
    for(i32 stack = 0; stack < _stacks; stack++) {
        if(! (stack == 0 || stack == stacks-1)) {
            //1st triangle
            coords.push_back(
                PointSpherical(_radius, ang_stack * (stack+1), ang_slice * (slice+1)));
            coords.push_back(
                PointSpherical(_radius, ang_stack * stack, ang_slice * (slice+1)));
            coords.push_back(
                PointSpherical(_radius, ang_stack * stack, ang_slice * slice));
        }
        //2nd triangle
        coords.push_back(
            PointSpherical(_radius, ang_stack * (stack+1), ang_slice * slice));
        coords.push_back(
            PointSpherical(_radius, ang_stack * (stack+1), ang_slice * (slice+1)));
        coords.push_back(
            PointSpherical(_radius, ang_stack * stack, ang_slice * slice));
    }
```

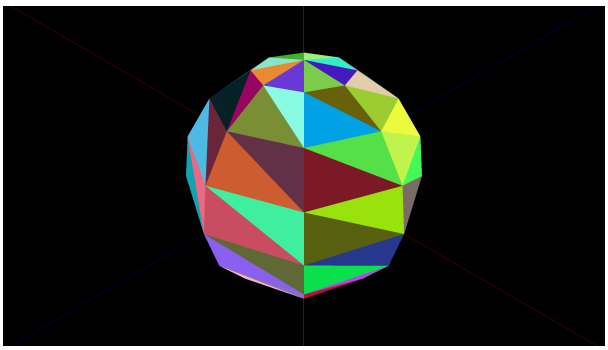

Figura 3.6: Esfera: raio:5 slices:8 stacks:8

3.4 Cone

Para desenhar um cone é necessário saber o seu raio (*_radius*), a sua altura (*_height*), o número de slices (*_slices*) e o número de stacks (*_stacks*).
Primeiramente é calculado o ângulo entre cada slice ((ang_slice)) e é definido um ponto que se encontra no topo do cone (*top_point*).

```
float ang_slice = 2 * M_PI / _slices;
Point top_point = Point(0, _height, 0);
```

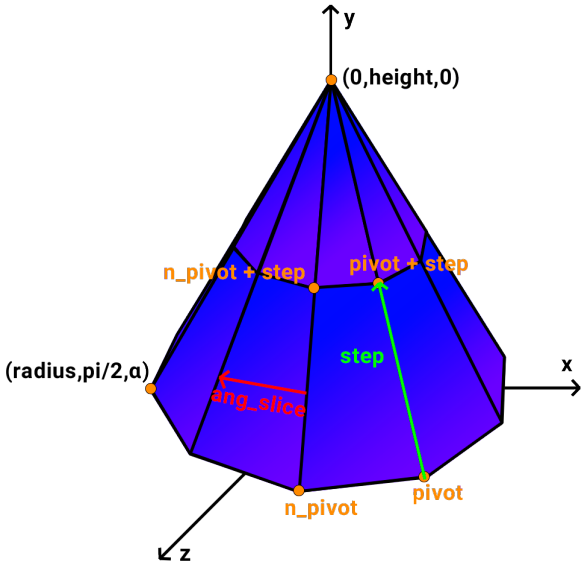

Figura 3.7: pontos e vetores no cone

Assim, os pontos na base do cone são da forma *(radius, π, a)*. Logo para desenhar a base do cone fazemos:

```
for(i32 slice = 0; slice < _slices; slice++) {
    Point base   = PointSpherical(_radius, M_PI/2, ang_slice *  slice);
    Point n_base = PointSpherical(_radius, M_PI/2, ang_slice * (slice+1));

    coords.push_back(n_base);
    coords.push_back(base);
    coords.push_back(PointSpherical(0, 0, 0));

}
```

Em seguida é calculado em cada ciclo um vetor que vai do *base* e do *n_base* até ao topo do cone mas que só tem o comprimento de uma stack:

```
Vector step   = Vector(  base, top_point) / _stacks;
Vector n_step = Vector(n_base, top_point) / _stacks;
```

Assim, para obter um ponto na lateral do cone podemos fazer *base + step * stack*. Visto que no topo do cone os triângulos também se iriam sobrepor uma solução semelhante à usada na esfera foi aplicada. Assim, o código final fica:

```
for(i32 slice = 0; slice < _slices; slice++) {
    Point base   = PointSpherical(_radius, M_PI/2, ang_slice *  slice);
    Point n_base = PointSpherical(_radius, M_PI/2, ang_slice * (slice+1));

    coords.push_back(n_base);
    coords.push_back(base);
    coords.push_back(Point(0, 0, 0));

```

```
9       Vector step   = Vector(  base, top_point) / _stacks;
10      Vector n_step = Vector(n_base, top_point) / _stacks;
11
12      for(i32 stack = 0; stack < _stacks; stack++) {
13          Point pivot   =    base +   step * stack;
14          Point n_pivot =  n_base + n_step * stack;
15
16          if(!(stack == _stacks -1)) {
17              coords.push_back(pivot);
18              coords.push_back(n_pivot + n_step);
19              coords.push_back(pivot + step);
20          }
21          coords.push_back(pivot);
22          coords.push_back(n_pivot);
23          coords.push_back(n_pivot + n_step);
24      }
25  }
```

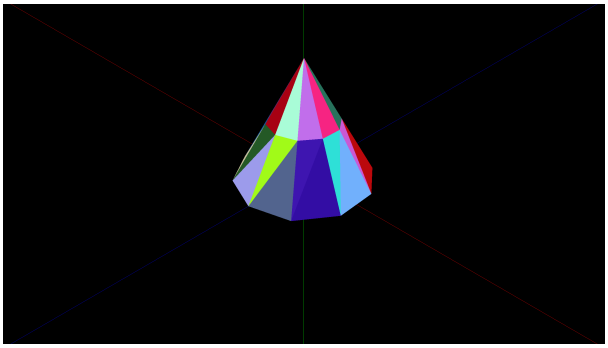

Figura 3.8: Cilindro: altura:5 raio:3 slices:10 stacks:2

3.5 Cilindro

Para desenhar o cilindro primeiro é definido um vetor que vai da base do cone ao topo do cone(top) e um vetor que tem a altura de uma stack do cilindro(step) e é calculado o ângulo que uma slice tem(ang_slice). Assim, para um ponto da aresta da base é dado por *(radius, π/2, ang_slice * slice)*.

```
1 Vector top = Vector(0, _height, 0);
2 Vector step = top / _stacks;
3 float ang_slice = 2 * M_PI / _slices;
```

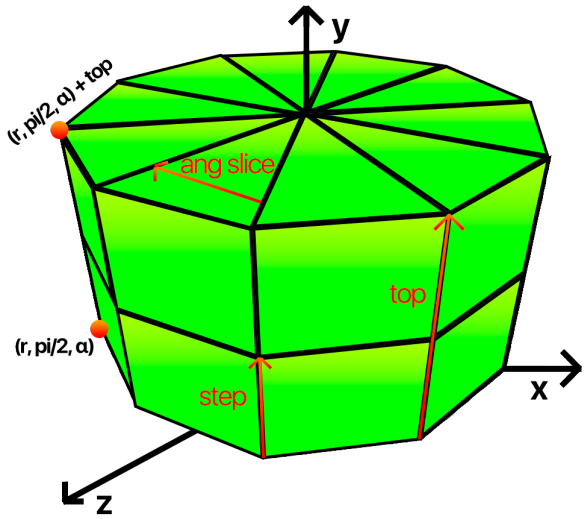

Figura 3.9: Vetores e ângulos dos cilindros

Assim para desenhar a base do cilindro e o topo do cilindro fazemos:

```
for(i32 slice = 0; slice < _slices; slice++) {
    PointSpherical central = PointSpherical(0, 0, 0);
    PointSpherical base    = PointSpherical(_radius, M_PI/2, ang_slice * slice);
    PointSpherical n_base  = PointSpherical(_radius, M_PI/2, ang_slice *(slice+1));
    //top & bottom
    coords.push_back(n_base);
    coords.push_back(base);
    coords.push_back(central);
    coords.push_back(central + top);
    coords.push_back(base + top);
    coords.push_back(n_base + top);
}
```

Para desenhar as laterais do cilindro coloca-se um ciclo for dentro do ciclo definido anteriormente de forma a percorrer as stacks do sólido. Assim, a versão final do código fica:

```
for(i32 slice = 0; slice < _slices; slice++) {
    PointSpherical central = PointSpherical(0, 0, 0);
    PointSpherical base    = PointSpherical(_radius, M_PI/2, ang_slice * slice);
    PointSpherical n_base  = PointSpherical(_radius, M_PI/2, ang_slice *(slice+1));
    //top & bottom
    coords.push_back(n_base);
    coords.push_back(base);
    coords.push_back(central);
    coords.push_back(central + top);
    coords.push_back(base + top);
    coords.push_back(n_base + top);

    //side
    for(i32 stack = 0; stack < _stacks; stack++) {
        Point pivot   =   base + step * stack;
        Point n_pivot = n_base + step * stack;

        coords.push_back(pivot);
        coords.push_back(n_pivot + step);
        coords.push_back(pivot + step);
        coords.push_back(pivot);
        coords.push_back(n_pivot);
        coords.push_back(n_pivot + step);
    }
}
```

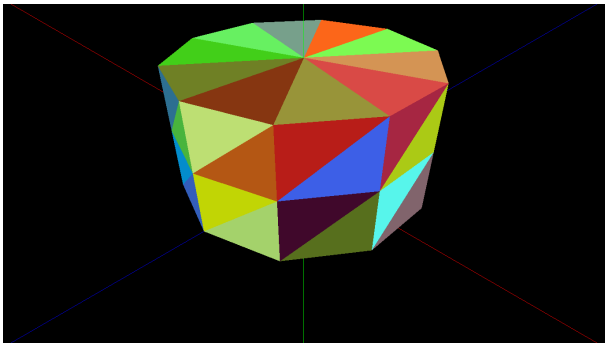

Figura 3.10: Cilindro: altura:5 raio:5 slices:10 stacks:2

Capítulo 4

Engine

Neste módulo foi desenvolvido um motor 3D simples.
Quando se executa o progrma gerado por defeito é aberta a scene guardada em *scenes/config.xml*. Se alguém ficehiro for passado como argumento esse será o selecionado.
Como ainda não existem sombras e luz as formas não são facilmente distinguíveis. De forma a resolver este problema decidimos atribuir a cada triângulo uma cor aleatória.

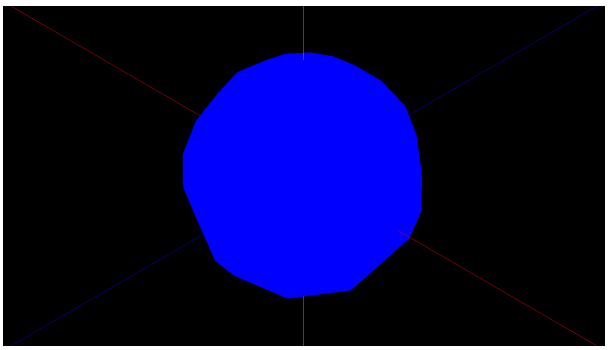

Figura 4.1: Esfera com cor sólida

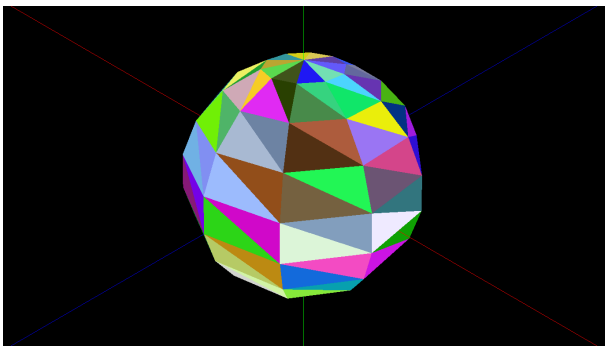

Figura 4.2: Esfera com várias cores por triângulo

4.1 3D Model loading

Para facilitar a leitura do ficheiro final gerado pelo gerador foi desenvolvida uma classe denominada de *Model*. Esta contem um vetor de pontos que são carregados do ficheiro anteriormente gerado e para o povoar apenas é necessário indicar o nome do ficheiro.

```
Model::Model(string fileName){
    float x, y, z;
    ifstream file(fileName.c_str());
    while (file >> x >> y >> z)
         points.push_back(Point(x, y, z));
}
```

Esta classe também contém um método para desenhar o modelo *draw_model* que chama um método para desenhar triângulos com cores aleatórias *draw_triangle*.

```
void draw_triangle(Point p1, Point p2, Point p3){
    float r = static_cast <float> (rand()) / static_cast <float> (RAND_MAX);
    float g = static_cast <float> (rand()) / static_cast <float> (RAND_MAX);
    float b = static_cast <float> (rand()) / static_cast <float> (RAND_MAX);
    glColor3f(r, g, b);
    glBegin(GL_TRIANGLES);
        glVertex3f(p1.x(), p1.y(), p1.z());
        glVertex3f(p2.x(), p2.y(), p2.z());
        glVertex3f(p3.x(), p3.y(), p3.z());
    glEnd();
}

void Model::draw_model() {
    for(i32 i = 0; i < points.size(); i += 3)
        draw_triangle(points[i], points[i+1], points[i+2]);
}
```

4.2 XML parsing

Para dar parsing ao XML utilizamos a biblioteca *rapidxml*. Nesta primeira versão do parser apenas obtemos quais são os ficheiros dos modelos a carregar contidos no xml. De forma a garantir que o ficheiro é lido apenas uma vez o carregamento é feito imediatamente quando o programa inicia e os ficheiros são carregas para a classe *Models* que contem um vetor de *Model* e o resultado é guardado num singleton.
Assim, para carregar todos os sólidos dentro do xml basta fazer:

```
models = Models(read_models(sceneName));
```

Para desenhar os modelos utilizamos o método *draw_models()*.

4.3 Camera

A camera consiste em dois pontos. Um ponto de coordenadas esféricas que representa a localização da camera no espaço e um ponto de coordenadas cartesianas que representa para onde a camera está a olhar.

```
PointSpherical _pl;
Point _center;
```

Para desenhar a camera converte se o ponto esférico num ponto cartesiano e utiliza-se a função *gluLookAt*.

```
Point c = Point(_pl);
glLoadIdentity();
gluLookAt(c.x(),c.y(),c.z(),
    _center.x(),_center.y(),_center.z(),
        0.0f,1.0f,0.0f);
```

Para mover a camera basta utilizar as funções definidas na classe *Point* e *PointSpherical* para alterar a posição da camera.

Capítulo 5

Conclusão

Com este trabalho prático podemos aplicar os conhecimentos lecionados até agora nas aulas da unidade curricular de computação gráfica permitindo assim que aprofundar os nossos conhecimentos de openGl. Como trabalho futuro gostaríamos de melhorar a camera de orbita que foi implementada no *engine*, introduzir mais cameras distintas e criar mais primitivas no *generator* como a torus, por exemplo.