

**Universidade do Minho**

Escola de Engenharia

Mestrado Integrado em Engenharia Informática

**Unidade Curricular de**

**Computação Gráfica**

Ano Lectivo de 2016/2017

|  |
| --- |
|  |
| Relatório do Trabalho Prático |
|  |
| 2016/2017 |

**Grupo**

Pedro Cunha A73958, José Silva A74601, Luís Fernandes A74748, João Coelho A74859

6 de março de 2017

Índice

[Introdução 2](#_Toc476601569)

[Fase 1 3](#_Toc476601570)

Primitivas gráficas ...…………………………………………………………………………3

Generator e engine …………………………………………………………………………13

[Conclusão 14](#_Toc476601571)

### Introdução

Com este trabalho pretende-se desenvolver um motor gerador de cenas baseadas em gráficos 3D, complementando-o com algumas figuras exemplo que mostrem o seu potencial. O trabalho será dividido em quatro fases, com diferentes fases de entrega.

Na primeira fase, o objetivo recai na criação de um programa gerador de ficheiros com informação sobre os vértices das figuras gráficas a serem desenhadas pelo motor. As primitivas gráficas requeridas são um plano, uma caixa, uma esfera e um cone.

### Fase 1

1. Primitivas gráficas
   * 1. Plano

std::vector<std::string> plane(float x, float y, float z, int div)

* x -> comprimento em x;
* y -> posição do plano em y;
* z -> comprimento em z;
* div -> número de divisões.

**for(int i = 0; i < div; i++) {**

**for(int j = 0; j < div; j++) {**

**v.push\_back(vertexString(x/2-x/div\*(i+1),**

**y, -z/2+z/div\*j));**

**v.push\_back(vertexString(x/2-x/div\*i,**

**y, -z/2+z/div\*(j+1)));**

**v.push\_back(vertexString(x/2-x/div\*i,**

**y, -z/2+z/div\*j));**

**v.push\_back(vertexString(x/2-x/div\*(i+1),**

**y, -z/2+z/div\*j));**

**v.push\_back(vertexString(x/2-x/div\*(i+1),**

**y, -z/2+z/div\*(j+1)));**

**v.push\_back(vertexString(x/2-x/div\*i,**

**y, -z/2+z/div\*(j+1)));**

**}**

**}**

5

4 6 x

z

1 3

2

Considere-se a figura acima, onde está ilustrado o caso de um plano com duas divisões (estão representadas apenas as duas colunas, falta mais uma linha). Olhando para o algoritmo, os 6 vértices gerados representam os dois triângulos de cada divisão. O primeiro trio representa, no caso da primeira divisão, os vértices 1;5;4 e o segundo 1;2;5, por esta ordem. Os vértices são gerados na ordem necessária para poderem ser visíveis no desenho. O algoritmo é executado até gerar os vértices em todas as colunas de cada linha.

* + 1. Caixa

std::vector<std::string> box(float x, float y, float z, int div)

* x -> comprimento em x;
* y -> altura;
* z -> comprimento em z;
* div -> número de divisões.

std::vector<std::string> v;

//top

v = plane(x, y/2, z, div);

//Usa a função plane para desenhar o topo da caixa.

for(int i = 0; i < div; i++) {

for(int j = 0; j < div; j++) {

//base

v.push\_back(vertexString(x/2-x/div\*(i+1), -y/2, -z/2+z/div\*j));

v.push\_back(vertexString(x/2-x/div\*i, -y/2, -z/2+z/div\*j));

v.push\_back(vertexString(x/2-x/div\*i, -y/2, -z/2+z/div\*(j+1)));

v.push\_back(vertexString(x/2-x/div\*(i+1), -y/2, -z/2+z/div\*j));

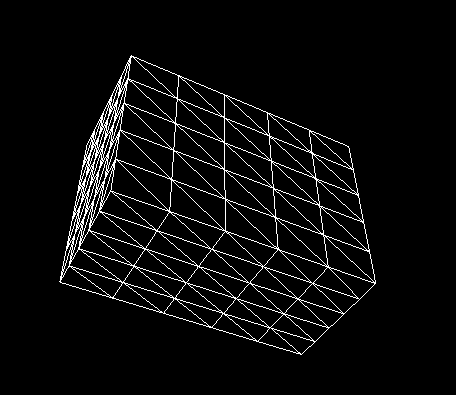
v.push\_back(vertexString(x/2-x/div\*i, -y/2, -z/2+z/div\*(j+1)));

v.push\_back(vertexString(x/2-x/div\*(i+1), -y/2, z/2+z/div\*(j+1)));

}

}

Está apenas a base representada, pois chega como exemplo de comparação com o topo desenhado usando a função plane. O que muda na base em relação ao topo é a ordem pela qual são gerados os vértices, isto porque é necessário que a base fique virada para baixo. Usando novamente a figura do plano, os vértices da primeira divisão, no caso da base, seriam na ordem 1;4;5 e 1;5;2. Relativamente a este exemplo, a única mudança nos restantes casos é que os vértices deixam de ser em XZ e passam a XY e YZ.



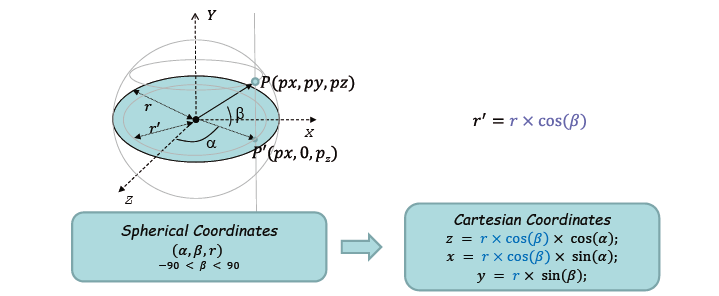
* + 1. Esfera

O desenho da esfera requer alguns parâmetros de input, entre eles o raio que a esfera irá tomar, o número de slices e o número de stacks.

std::vector<std::string> sphere(float radius, int slices, int stacks)

* radius -> raio da esfera;
* slices -> número de divisões verticais da figura (ao longo do eixo dos yy);
* stacks -> número de divisões horizontais da figura (ao redor do eixo dos yy).

Passando ao algoritmo usado, foi inicialmente planeado o uso da conversão de coordenadas esféricas para cartesianas, como demonstra a seguinte figura:



for (int i = 0; i < stacks; i++) {

1º Ciclo que vai incrementando o i quando passamos para a próxima stack a desenhar.

for (int j = 0; j < slices; j++) {

2º Ciclo que vai incrementando o j quando passamos para a próxima slice a desenhar.

Este ciclo encontra-se dentro do anterior, visto que vamos desenhar todas as slices necessárias a cada camada da stack.

if (i != stacks - 1) {

fi = (i + 1) \* M\_PI/stacks;

teta = (j + 1) \* 2 \* M\_PI/slices;

v.push\_back(vertexString(radius \* cos(teta)\*sin(fi),

radius \* cos(fi),

radius \* sin(teta)\*sin(fi)));

teta = j \* 2 \* M\_PI/slices;

v.push\_back(vertexString(radius \* cos(teta)\*sin(fi),

radius \* cos(fi),

radius \* sin(teta)\*sin(fi)));

fi = i \* 2 \* M\_PI/slices;

v.push\_back(vertexString(radius \* cos(teta)\*sin(fi),

radius \* cos(fi),

radius \* sin(teta)\*sin(fi)));

}

if (i != 0) {

fi = i \* M\_PI/stacks;

teta = j \* 2 \* M\_PI/slices;

v.push\_back(vertexString(radius\*cos(teta)\*sin(fi),

radius \* cos(fi),

radius \* sin(teta)\*sin(fi)));

teta = (j + 1) \* 2 \* M\_PI/slices;

v.push\_back(vertexString(radius \* cos(teta)\*sin(fi),

radius \* cos(fi),

radius \* sin(teta) \* sin(fi)));

fi = (i + 1) \* M\_PI/stacks;

v.push\_back(vertexString(radius \* cos(teta)\*sin(fi),

radius \* cos(fi),

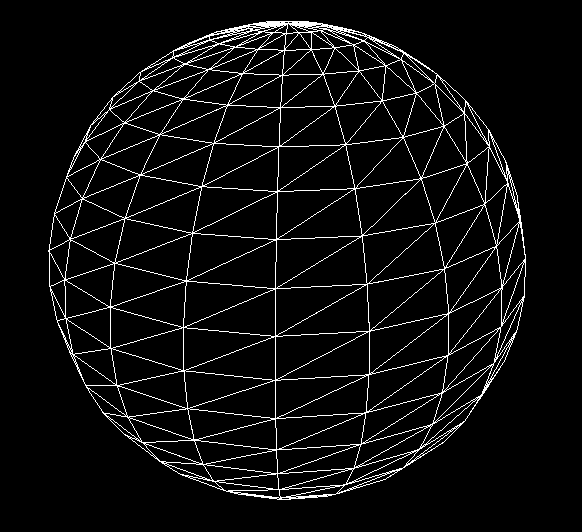
radius \* sin(teta) \* sin(fi)));

}

Dentro dos ciclos anteriormente referidos encontram-se 2 blocos if. Como é possível verificar, ambos são executados, a menos que a camada da stack a desenhar seja a primeira ou a última, isto porque na primeira e na última camada, cada slice vai ser constituída apenas por um triângulo e não por dois, como acontece em todas as outras camadas. Dito isto, caso estejamos a desenhar a primeira camada, apenas o primeiro bloco if é executado, colocando as variáveis float fi e float teta com os ângulos necessários para o desenho do triângulo, dando aqui uso às primitivas referidas anteriormente de conversão de coordenadas esféricas em coordenadas cartesianas.

Da mesma forma, é executado apenas o 2º bloco if caso a camada da stack seja a última. Quando a camada a desenhar não é a primeira nem a última, ambos os blocos são executados e, desta forma, cada um deles desenha um triângulo, os quais juntos vão dar forma ao retângulo que forma a slice dessa camada da stack.

Como nas outras formas desenhadas, é de salientar que a ordem pela qual são desenhados os vértices dos triângulos é relevante, visto que é assim que garantimos que o conseguimos visualizar do ponto de vista pretendido.



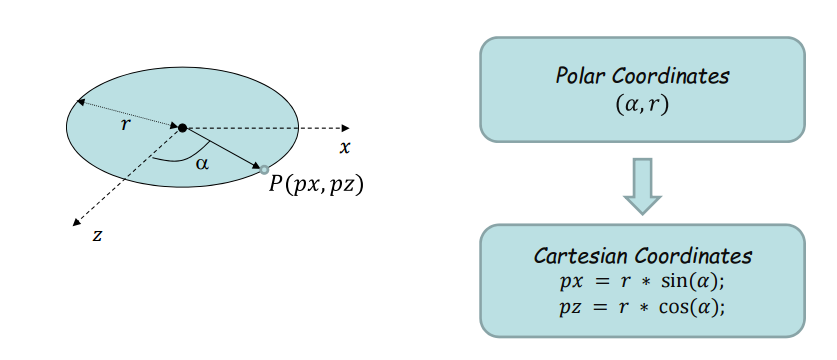
* + 1. Cone

Tal como constava no enunciado, o desenho do cone requer o fornecimento do raio da base, da altura e dos números de slices e stacks. Para diversificação das figuras passíveis de serem desenhadas pelo modelo, decidiu-se fornecer também o raio do topo, permitindo assim o desenho de um cilindro ou de um “cone cortado”.

std::vector<std::string> cylinder(float radB, float radT, float height, int slices, int stacks

* radB -> raio da base da figura;
* radT -> raio do topo da figura – quando é 0 temos um cone, quando é igual a radB temos um cilindro e nos demais casos temos um “cone cortado”.
* height -> altura da figura;
* slices -> número de divisões verticais da figura (ao longo do eixo dos yy);
* stacks -> número de divisões horizontais da figura (ao redor do eixo dos yy).

Em relação ao algoritmo utilizado, inicialmente desenhou-se a base da primitiva gráfica. Para isso, recorremos à conversão de coordenadas polares em coordenadas cartesianas, tal como surge na figura seguinte:

O alpha definido toma o valor da divisão do ângulo total de uma circunferência (2Π) pelo número de slices:

**float alpha = (float)2\* M\_PI / slices;**

Assim, em função do ponto que estamos a desenhar, o ângulo entre a linha do raio e o eixo dos zz (angle no código e alfa da imagem) varia entre 0 e slices \* alpha.

**std::vector<std::string> v;**

**for (j = 1; j <= slices; j++) { //ciclo 1**

**v.push\_back(vertexString (sin(alpha\*(j-1))\*radB, 0.0f, cos(alpha\*(j-1))\*radB));**

**angle = alpha \* j;**

**v.push\_back(vertexString (0.0f, 0.0f, 0.0f));**

**v.push\_back(vertexString (sin(angle)\*radB, 0.0f, cos(angle)\*radB));**

Isto permite-nos desenhar a base. Note-se que a ordem com que são desenhados os vértices é relevante. Usando a estratégia da regra da mão direita, ordenou-se desta forma os vértices, conseguindo que a face da base esteja a apontar para fora e permitindo-nos ver a coloração dos triângulos.

Posteriormente, passou-se ao desenho das faces laterais da figura. Aqui há diferenças consoante a figura seja um cone, um cilindro ou um “cone cortado”, porém essa diferença traduz-se na variação de uma única linha de código (assinalada em baixo por **∇**). Essa linha representa o comprimento do raio à medida que se sobe nas *stacks.* Em termos da representação dos pontos, a subida na *stack* é traduzida pelo aumento de **h** unidades no valor da coordenada y dos pontos**:**

**float h = (float) height / stacks;**

Quando temos um cilindro, como os raios da base e do topo são iguais, trabalha-se sempre com o valor de **radB** no desenho dos triângulos das faces laterais – da base até ao topo – mas no caso de a figura ser um cone, por cada *stack* que se sobe, o raio utilizado no cálculo das coordenadas dos pontos decresce **radB/*stacks*** unidades – variável **lvl**. Assim, quando se atinge o número total de *stacks* temos o valor 0. Quando se desenha uma figura intermédia entre o cilindro e o cone, que intitulamos de “cone cortado”, ao percorrer cada *stack,* a redução do valor do raio a usar nos cálculos é apenas (**radB – radT**)**/*stacks***, portanto não se atinge o valor 0 e não temos o vértice superior do cone, mas sim um círculo com raio **radT** no topo.

**prev\_lvl = lvl;**

**for(int i = 1; i <= stacks; i++){ //ciclo 2**

**∇ lvl = radB - (float) (radB - radT) \* i/stacks;**

**v.push\_back(vertexString((prev\_lvl\*sin(alpha\*(j-1)), h\*(i-1), prev\_lvl\*cos(alpha\*(j-1)));**

**v.push\_back(vertexString(prev\_lvl\*sin(angle), h\*(i-1), prev\_lvl\*cos(angle));**

**v.push\_back(vertexString(lvl\*sin(alpha\*(j-1)), h\*i, lvl\*cos(alpha\*(j-1)));**

**v.push\_back(vertexString(lvl\*sin(alpha\*(j-1)), h\*i, lvl\*cos(alpha\*(j-1)));**

**v.push\_back(vertexString(prev\_lvl\*sin(angle), h\*(i-1), prev\_lvl\*cos(angle));**

**v.push\_back(vertexString(lvl\*sin(angle), h\*i, lvl\*cos(angle));**

**prev\_lvl = lvl;**

**}**

Nota para a variável **prev\_lvl**, que toma o valor de **lvl** sempre que avançamos de *slice,* isto é, sempre que completamos um ciclo 2 e seguimos para a iteração seguinte do ciclo 1. Em termos de significado prático, **prev\_lvl** representa o valor do raio utilizado no cálculo das coordenadas dos pontos na *stack* anterior.

Quando a figura não é um cone, é também necessário desenhar o topo. Eis que surge assim o terceiro ciclo:

**if (radT) { //ciclo 3**

**for (j = 1; j <= slices; j++) {**

**angle = alpha \* j;**

**v.push\_back(vertexString (sin(angle)\*radT, height, cos(angle)\*radT);**

**angle = alpha \* (j - 1);**

**v.push\_back(vertexString (0.0f, height, 0.0f);**

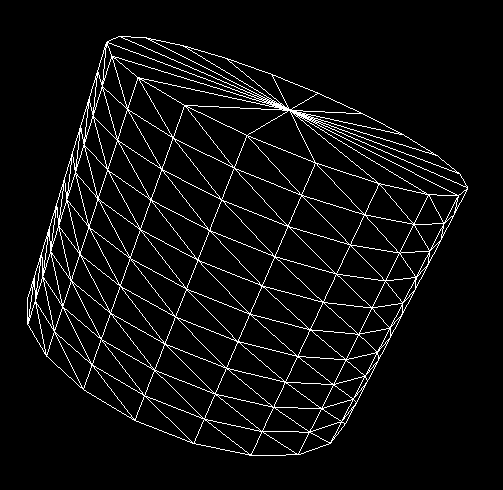
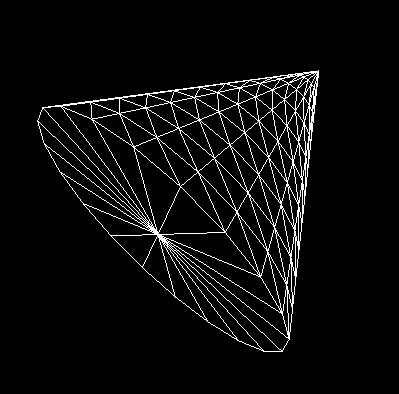
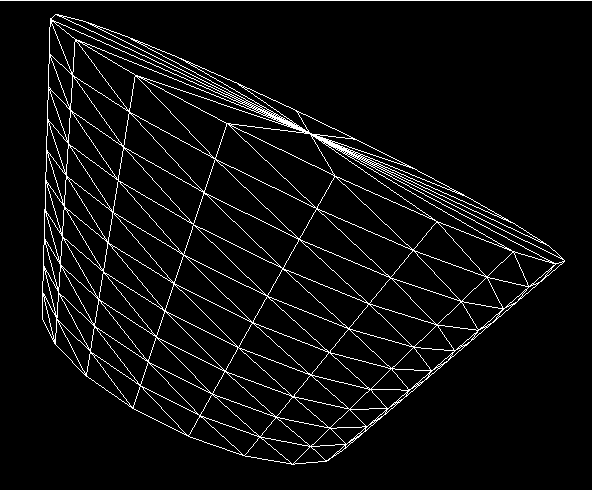
**v.push\_back(vertexString (sin(angle)\*radT, height, cos(angle)\*radT);**

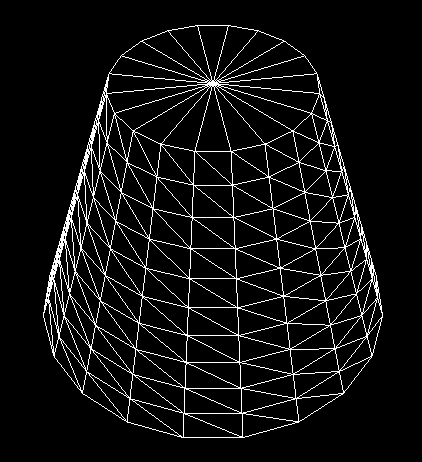
**}**

**}**

Destaca-se a ordem oposta no desenho dos vértices, quando se compara com o desenho dos triângulos da base, porque naturalmente o topo fica voltado no sentido oposto. De resto, o processo é equivalente ao utilizado no desenho da base: de *slice* em *slice,* o **alpha** vai crescendo e é utilizado para calcular as posições dos pontos dos triângulos.

Com esta função podemos desenhar qualquer uma das seguintes 4 figuras:





* + 1. Toro

O desenho do toro requer o fornecimento da distância entre o centro e o centro do tubo, da distância entre o centro e o início do tubo e os números de sides e rings.

**std::vector<std::string> torus(float inner, float outer, int sides, int rings)**

* inner -> raio interior - entre o centro e o início do tubo;
* outer -> raio exterior – entre o centro e o centro do tubo;
* sides -> número de lados de cada secção radial do toro;
* rings -> número de secções radiais (“anéis”) do toro.

Relativamente ao algoritmo usado, serão percorridos todos os “anéis” do toro, calculando as coordenadas de todos os pontos respetivos a um anel aquando da passagem pelo mesmo.

Envolve o cálculo de um alpha:

**float alpha = (float) 2 \* M\_PI / rings;**

que representa o aumento, por cada anel que se percorre, no ângulo entre o eixo dos zz e a linha do raio que interseta a extremidade do anel.

Porém, neste caso necessita-se de um segundo ângulo, beta:

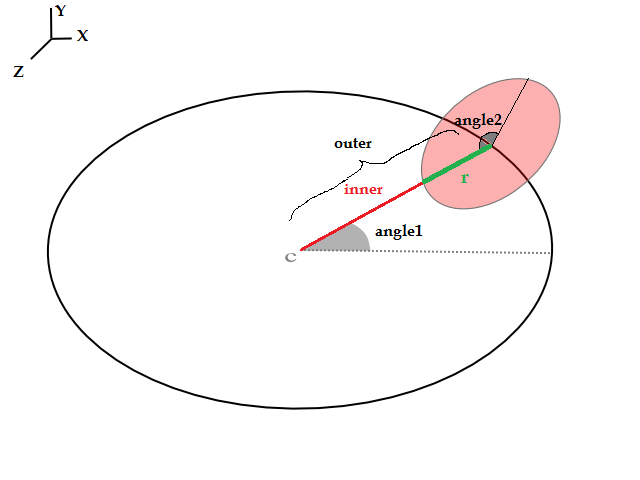
**float beta = (float) 2 \* M\_PI / sides;**

que se trata do acréscimo, por cada *side* que se avança, do ângulo entre a horizontal do tubo e a linha do raio que marca o *side* atual.

Para além destes ângulos, calculou-se também o valor do raio do tubo:

**float r = outer - inner;**

O conjunto destas variáveis calculadas é usado, a par das variáveis recebidas na chamada da função, na construção das várias secções tubulares. O processo é simples:

* **angle1** representará o “anel” que estamos a construir - em termos práticos representa quantos **alphas** se percorreu – e, por cada **angle1,** o **angle2** oscilará entre 0 e *sides* \* **beta**, pelo que se consegue assim desenhar os pontos sobre as linhas de divisão de todos *sides.* No excerto de código abaixo, o ciclo 1 representa a variação do **angle1** e o ciclo 2 a variação do **angle2**.
* Cada *side* é formado por 2 triângulos, logo são definidos 6 pontos. A coordenada X é definida pelo produto do seno do **angle1** pelo valor da distância entre o centro do toro e a projeção do ponto que estamos a definir no plano horizontal do centro. Analisando a imagem acima, podemos observar que essa distância variará entre **outer – r** e **outer + r**. A coordenada Z é calculada da mesma forma, porém utiliza-se o cosseno do **angle1.** Durante este processo, nota para o uso do cosseno do **angle2** para calcular o valor a somar/subtrair ao **outer** e do seno para calcular a coordenada Y dos pontos, em ambos os casos multiplicando por **r**. Passando por todos os *sides,* termina-se o desenho de todos os pontos de um *ring.*
* Mais uma vez, a ordem dos vértices é importante. Como, no toro, todos os triângulos têm de ficar voltados no mesmo sentido, mantém-se a ordem com que se desenham os triângulos, independentemente do *side* e *ring* em que se está. Isto resulta num algoritmo executado com apenas dois ciclos.

**std::vector<std::string> v;**

**for (int i = 1; i <= rings; i++) { // ciclo 1**

**for (int j = 1; j <= sides; j++) { // ciclo 2**

**angle1 = alpha \* (i - 1);**

**angle2 = beta \* (j - 1);**

**v.push\_back(vertexString(sin(angle1)\*(outer - r\*cos(angle2)),**

**r \* sin(angle2),**

**cos(angle1)\*(outer - r\*cos(angle2))));**

**angle2 = beta \* j;**

**v.push\_back(vertexString(sin(angle1)\*(outer - r\*cos(angle2)),**

**r \* sin(angle2),**

**cos(angle1)\*(outer - r\*cos(angle2))));**

**angle2 = beta \* (j-1);**

**angle1 = alpha \* i;**

**v.push\_back(vertexString(sin(angle1)\*(outer - r\*cos(angle2)),**

**r \* sin(angle2),**

**cos(angle1)\*(outer - r\*cos(angle2))));**

**v.push\_back(vertexString(sin(angle1)\*(outer - r\*cos(angle2)),**

**r \* sin(angle2),**

**cos(angle1)\*(outer - r\*cos(angle2))));**

**angle1 = alpha \* (i - 1);**

**angle2 = beta \* j;**

**v.push\_back(vertexString(sin(angle1)\*(outer - r\*cos(angle2)),**

**r \* sin(angle2),**

**cos(angle1)\*(outer - r\*cos(angle2))));**

**angle1 = alpha \* i;**

**v.push\_back(vertexString(sin(angle1)\*(outer - r\*cos(angle2)),**

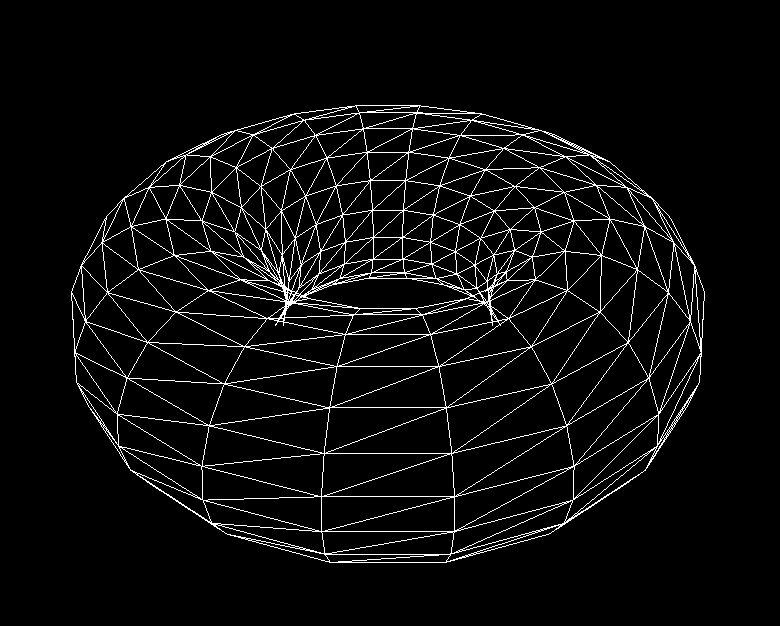
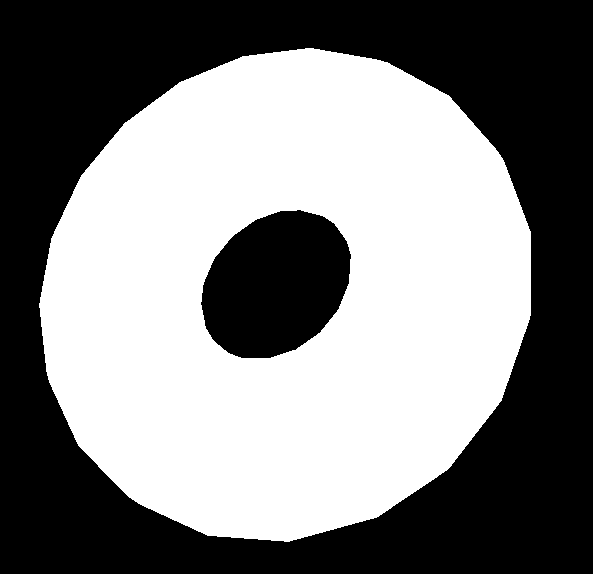
**r \* sin(angle2),**

**cos(angle1)\*(outer - r\*cos(angle2))));**

**}**

**}**

**return v;**

 Por fim, o toro resultante:

* + 1. Rubi

Esta figura, que se designou como rubi, trata-se de desenhar, para a mesma base, um “cone cortado” e um cone no sentido oposto. A função receberá, portanto:

**vector<string> drawRubi(int rb, int rt, int hb, int ht, int slices, int stacks)**

* rb - > raio da base, comum ao cone e ao “cone cortado”;
* rt -> raio do topo do rubi, isto é, do “cone cortado”;
* hb -> altura do cone;
* ht -> altura do “cone cortado”;
* slices -> número de divisões verticais da figura (ao longo do eixo dos yy);
* stacks -> número de divisões horizontais da figura (ao redor do eixo dos yy).

O algoritmo é muito simples, já que essencialmente reutiliza o código da função criada para gerar o cone.

**drawCone(rb, rt, ht, slices, stacks);**

A diferença está unicamente no sentido da coordenada Y dos pontos do cone da base, que passou a negativo para que o cone cresça no sentido negativo. As variáveis angle, alpha, h, lvl e prev\_lvl possuem o mesmo significado comparativamente à função exposta na secção do cone.

**std::vector<std::string> v;**

**for (j = 1; j <= slices; j++) {**

**angle = alpha \* j;**

**prev\_lvl = rb;**

**for (i = 1; i <= stacks; i++) {**

**lvl = rb - (float)rb\*i / stacks;**

**v.push\_back(vertexString (prev\_lvl \* sin(angle), -h\*(i - 1), prev\_lvl \* cos(angle));**

**v.push\_back(vertexString (prev\_lvl \* sin(alpha\*(j - 1)), -h\*(i - 1), prev\_lvl \* cos(alpha\*(j - 1)));**

**v.push\_back(vertexString (lvl \* sin(alpha\*(j - 1)), -h \* i, lvl \* cos(alpha \* (j - 1)));**

**v.push\_back(vertexString (lvl \* sin(alpha\*(j - 1)), -h \* i, lvl \* cos(alpha \* (j - 1)));**

**v.push\_back(vertexString (lvl \* sin(angle), -h\*i, lvl \* cos(angle));**

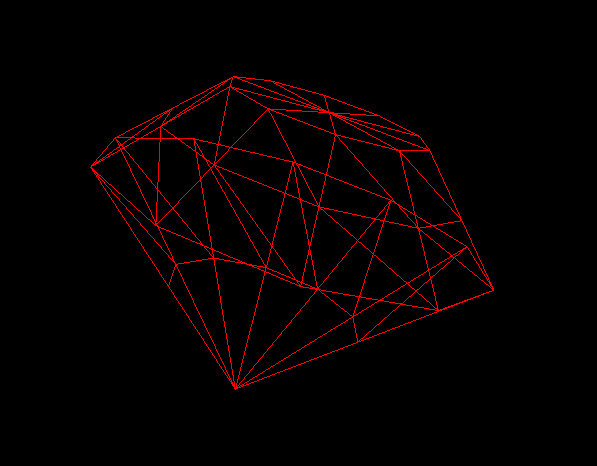
**v.push\_back(vertexString (prev\_lvl \* sin(angle), -h\*(i - 1), prev\_lvl \* cos(angle));**

**prev\_lvl = lvl;**

**}**

**}**

Para que se perceba o resultado, eis a figura resultante:



1. Generator e engine

Algumas notas sobre o funcionamento dos executáveis relativos ao gerador de pontos das figuras e ao motor que lê esses pontos e desenha as figuras:

* Generator

bool verifyInput(char \*\*input) {

- Verifica se o input tem a forma/argumentos devidos.

std::vector<std::string> getVertexes(char \*\*input) {

* Cria os vértices da figura pretendida, passada no input com os argumentos, após a validação executada na função previamente explicada. Para isto chama as funções respetivas da criação dos vértices do ficheiro shapes.cpp, funções explicadas na secção das primitivas gráficas.
* Engine

De salientar que foi implementada a Explorer Cam (GLUT\_KEY\_UP, GLUT\_KEY\_DOWN, GLUT\_KEY\_LEFT e GLUT\_KEY\_RIGHT), bem como a possibilidade de alterar entre Fill, Line e Point nas figuras (GLUT\_KEY\_F1).

Por último, mas não menos importante, o parser de xml utilizado foi o tinyxml2.

### Conclusão

Esta primeira fase do projeto permitiu o ganho de alguma prática e destreza na construção de primitivas gráficas e no controlo do *camera motion,* bem como no manuseamento de ferramentas como o CMake.

As aplicações criadas para o *generator* e para o *engine* foram uma mais valia no processo de aprendizagem de C++, que durará ao longo das restantes fases do projeto.

Relativamente à satisfação do grupo quanto ao trabalho realizado, pela construção de figuras adicionais pode-se subentender o grande interesse que o trabalho despertou nos membros.