



UNIVERSIDADE DO MINHO
LICENCIATURA EM ENGENHARIA INFORMÁTICA

COMPUTAÇÃO GRÁFICA

Trabalho Prático - Fase 1

Grupo 33



Bohdan Malanka
a93300



Diogo Rebelo
a93278



Henrique Alvelos
a93316



Lídia Sousa
a93205

8 de junho de 2022

Conteúdo

| | | |
|----------|----------------------------------|-----------|
| 1 | Introdução | 3 |
| 2 | Descrição do Problema | 3 |
| 3 | Estrutura das aplicações | 4 |
| 4 | Gerador | 6 |
| 4.1 | Ficheiro 3D | 6 |
| 4.2 | Primitivas | 6 |
| 4.2.1 | Plano | 6 |
| 4.2.2 | Caixa | 6 |
| 4.2.3 | Cone | 7 |
| 4.2.4 | Esfera | 7 |
| 4.2.5 | Cilindro | 8 |
| 4.2.6 | Toro | 8 |
| 5 | Motor | 10 |
| 5.1 | Extras | 10 |
| 5.1.1 | Movimentação da câmara | 10 |
| 6 | Testes | 12 |
| 6.1 | Cone | 12 |
| 6.2 | Esfera | 13 |
| 6.3 | Caixa | 14 |
| 6.4 | Toro | 15 |
| 6.5 | Cilindro | 16 |
| 6.6 | Figuras sobrepostas | 17 |
| 7 | Conclusão | 19 |

Lista de Figuras

| | | |
|----|---|----|
| 1 | Estrutura da Aplicação: Engine | 4 |
| 2 | Estrutura da Aplicação: Gerador | 4 |
| 3 | Estrutura da Aplicação: Models | 5 |
| 4 | Exemplo de um ficheiro .3d | 6 |
| 5 | Variáveis representadas graficamente no Toro ¹ | 8 |
| 6 | Toro dividido em <i>slices</i> e <i>stacks</i> ¹ | 9 |
| 7 | Estruturas de dados Camera e World | 10 |
| 8 | Comando: generator cone 1 2 4 3 cone.3d | 12 |
| 9 | Comando: generator sphere 1 10 10 sphere.3d | 13 |
| 10 | Comando: generator box 2 3 box.3d | 14 |
| 11 | Comando: generator torus 1 3 10 10 torus.3d | 15 |
| 12 | Comando: generator cylinder 1 3 15 cylinder.3d | 16 |
| 13 | Sobreposição das figuras anteriores | 17 |
| 14 | Comando: Sobreposição das figuras anteriores | 18 |

Listings

| | | |
|---|---------------------------------|---|
| 1 | Point Structure | 4 |
| 2 | Parametização do Toro | 9 |

1 Introdução

O presente documento é alusivo à primeira fase do projeto prático desenvolvido com recurso à linguagem de programação C++, no âmbito da Unidade Curricular de Computação Gráfica que integra a Licenciatura em Engenharia Informática da Universidade do Minho. Este projeto encontra-se dividido em quatro fases de trabalho, cada uma com uma data de entrega específica. Esta divisão em fases, pretende fomentar uma simplificação e organização do trabalho, contribuindo para a sua melhor compreensão.

Pretende-se, assim, que o relatório sirva de suporte ao trabalho realizado para esta fase, mais propriamente, dando uma explicação e elucidando o conjunto de decisões tomadas ao longo da construção de todo o código fonte e descrevendo a estratégia utilizada para a concretização dos principais objetivos propostos, que surgem a seguir:

- Compreender a utilização do *OpenGL*, recorrendo à biblioteca *GLUT*, para a construção de modelos 3D;
- Aprofundar temas alusivos à produção destes modelos 3D, nomeadamente, em relação a transformações geométricas, curvas, superfícies, iluminação, texturas e modo de construção geométrico básico;
- Relacionar todo o conceito de construir modelos 3D com o auxílio da criação de ficheiros que guardam informação relevante nesse âmbito;
- Relacionar aspetos mais teóricos com a sua aplicação a nível mais prático.

Naturalmente, é indispensável que o conjunto de objetivos supracitados seja concretizado com sucesso e, para isso, o formato do relatório está organizado de acordo com uma descrição do problema inicial, seguindo-se o conjunto de aspetos relevantes em relação ao *Generator* e chegando aos aspetos primordiais sobre o *Engine*. O grupo decidiu incluir também uma secção direcionada para a descrição das funcionalidades adicionais.

2 Descrição do Problema

Após a leitura atenta do respetivo enunciado de trabalho para a respetiva fase inicial, compreende-se que esta primeira fase visa a construção de duas entidades funcionalmente distintas (dois programas distintos): um gerador - que gera um ficheiro com o conjunto de pontos do modelo em questão - e um motor - que lê um ficheiro de configuração no formato XML, representando assim graficamente os modelos. A primeira tabela reúne a explicação dos requisitos para o desenho das formas geométricas.

| Forma Geométrica | Parâmetros |
|------------------|--|
| Plano | length, divisions, .3d filename |
| Caixa | units, grid, .3d filename |
| Cone | radius, slices, stacks, .3d filename |
| Esfera | radius, height, slices, stacks, .3d filename |
| Cylinder | radius, height, slices, .3d filename |
| Toro | in_radius, out_radius, slices, tacks, .3d filename |

Tabela 1: Parâmetros a fornecer para gerar cada modelo geométrico 3D.

Em relação ao Motor, este lê os modelos do ficheiro XML, armazena os vértices dos modelos em memória e desenha as primitivas relativas aos modelos, sempre utilizando triângulos.

No contexto do problema, tornou-se relevante definir uma estrutura **Point**, importante no auxílio da representação dos vértices num plano 3D. A constituição da mesma surge a seguir:

Listing 1: Point Structure

```
struct Point {  
    float x;  
    float y;  
    float z;  
};
```

3 Estrutura das aplicações

A estrutura da aplicação é intuitiva. Decidimos organizar cada fase numa diretoria diferente, assim como, cada entidade tem a sua respetiva diretoria com código fonte.

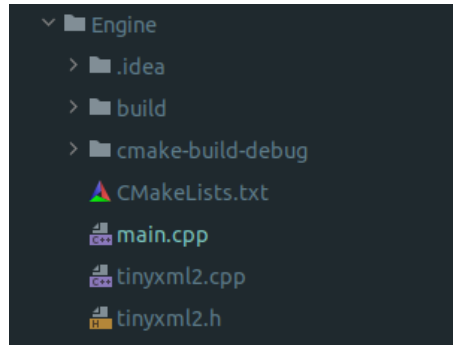


Figura 1: Estrutura da Aplicação: Engine

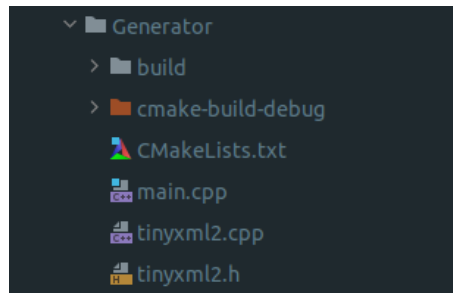


Figura 2: Estrutura da Aplicação: Gerador

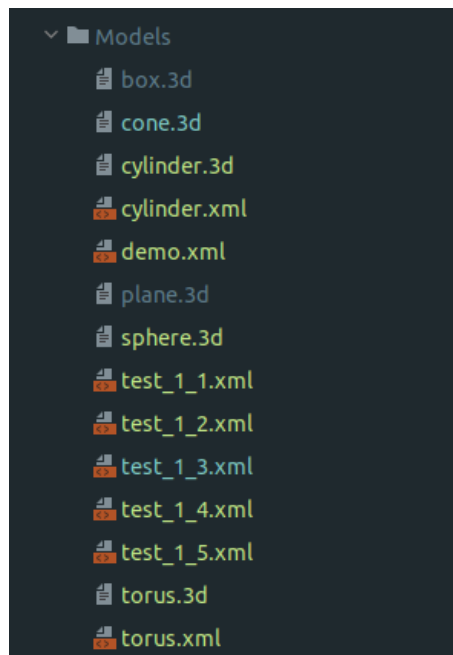


Figura 3: Estrutura da Aplicação: Models

De um modo geral, a aplicação é construída pelas seguintes componentes:

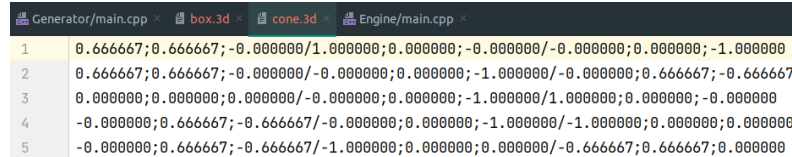
- **Gerador:** contém o conjunto de funções e estruturas responsáveis por gerar o ficheiro com o conjunto de pontos de cada modelo, com a extensão `.3d`;
- **Motor:** contém o conjunto de funções e estruturas responsáveis por ler o ficheiro de configuração XML e representar graficamente cada modelo;
- **Models:** diretoria onde os ficheiros `.3d` e os ficheiros XML ficam guardados.

4 Gerador

O gerador cria os ficheiros .3d com as primitivas para serem usados pelo motor.

4.1 Ficheiro 3D

Os ficheiros .3d contêm todos os vértices (*struct Point*) de uma primitiva. A estrutura de um ficheiro .3d tem uma formatação que consiste em cada linha corresponder a um triângulo, os pontos do triângulo estão separados pelo '/' pela ordem correta e as coordenadas separadas por ';'. Esta formatação respeita a regra da mão direita.



```
Generator/main.cpp x box.3d x cone.3d x Engine/main.cpp x
1 0.666667;0.666667;-0.000000/1.000000;0.000000;-0.000000/-0.000000;0.000000;-1.000000
2 0.666667;0.666667;-0.000000/-0.000000;0.000000;-1.000000/-0.000000;0.666667;-0.666667
3 0.000000;0.000000;0.000000/-0.000000;0.000000;-1.000000/1.000000;0.000000;-0.000000
4 -0.000000;0.666667;-0.666667/-0.000000;0.000000;-1.000000/-1.000000;0.000000;0.000000
5 -0.000000;0.666667;-0.666667/-1.000000;0.000000;0.000000/-0.666667;0.666667;0.000000
```

Figura 4: Exemplo de um ficheiro .3d

4.2 Primitivas

Além das primitivas requisitadas (plano, caixa, cone, esfera) decidimos implementar também o cilindro e o toro.

4.2.1 Plano

Para a construção do plano foi usada a função *createPlane* que, ao receber como parâmetros o comprimento, as divisões e o ficheiro .3d, vai criar um plano quadrado no plano XZ, guardando os pontos gerados no respetivo ficheiro.

Desta forma, para gerar esses pontos multiplicamos o valor atual da coordenada com a divisão do comprimento pelo número de divisões. Assim, evitamos os possíveis erros que se possam suceder de somas sucessivas. Começando, assim, na divisão $(-divisions/2)$ e incrementando este valor em uma unidade por cada iteração até $(divisions/2)$, percorrendo assim todas as divisões da face do plano. A Coordenada Y aqui será sempre 0.

Nesse sentido, percorrendo o eixo dos Z's e X's definimos 4 pontos por divisão:

```
p1 = x1, 0, z1;
p2 = x2, 0, z1;
p3 = x2, 0, z2;
p4 = x1, 0, z2;
```

Onde

```
x1 = axeX * length;
x2 = (axeX + 1) * length;
z1 = axeZ * length;
z2 = (axeZ + 1) * length;
```

Deste modo, para desenhar o plano, seguindo a regra da mão direita, a orientação dos pontos no plano é p2, p1 e p4 para um triângulo e p4, p3 e p2 para outro.

4.2.2 Caixa

Para esta primitiva foi opção utilizar a função que cria o plano e recorrer a translações e rotações necessárias para construir a caixa. No entanto, para a concepção da mesma a estratégia foi pegar num ponto da extremidade de uma face e, através de 2 vetores, construir cada face aproveitando pontos simétricos para faces simétricas. A escolha deste ponto é aleatória, isto porque se podia escolher qualquer ponto desta face, apenas os vetores mudariam em função do mesmo.

Inicialmente construíram-se as faces laterais - um ciclo com 2 iterações que representam cada um dos lados da caixa. Pensando na face paralela ao eixo xy e no ponto, vemos que os vetores que representam esta face são (1,0,0) e (0,1,0). Quanto à face paralela ao eixo yz, os vetores serão (0,0,-1) e (0,1,0). De seguida para a face da base e do topo (faces simétricas) começamos pelo ponto inferior esquerdo da face superior. Quanto aos vetores e pensando num plano cartesiano, a face "cresce" em relação ao eixo x e "diminui" em relação ao eixo z, daí que os vetores sejam (1,0,0) e (0,0,-1). Em relação à *grid*, decidiu-se guardar a primeira posição de cada linha para que, depois de a preencher, se consiga ir para a próxima.

4.2.3 Cone

Para a construção de um cone, foi usada a função *createCone* que ao receber como parâmetros o raio (*radius*) e altura (*height*), *stacks*(divisões horizontais), *slices*(divisões verticais) e o ficheiro .3, vai gerar os pontos desse cone e guardar no respetivo ficheiro. Deste modo, para representar o cone, iteramos para cada *Stacks* iteramos os *Slices* e vamos construindo da base até ao topo do cone. Assim, é necessário saber em cada *stack* três fatores:

- a sua altura (*stackHeight*)
- o seu ângulo de rotação para cada slice (*alpha*)
- o seu raio (*stackRadius*)

Quanto a **stackHeight** de cada *stack*, o cálculo é:

$$stackHeight = height / stacks$$

Relativamente ao ângulo **alpha**, apenas necessitamos de saber o número de *slices*, assim:

$$\alpha = (2 * \pi) / slices$$

Para o cálculo do **stackRadius** é necessário ter em conta o princípio matemático da semelhança de triângulos, já que o triângulo formado pelo **radius** e **height** do cone, com os triângulos formados em cada *stack* a partir da **stackHeight** são triângulos semelhantes visto que possuem os lados proporcionais e os ângulos congruentes. Assim:

$$stackRadius = radius * (height - (stackHeight * i)) / height$$

4.2.4 Esfera

Para construir uma esfera utilizamos coordenadas esféricas que consistem no conjunto de 3 valores pertencentes ao sistema esférico de coordenadas que permitem a localização de um ponto num qualquer espaço de formato esférico. Estas coordenadas definem-se através dos ângulos α e β e do raio (*radius* no contexto da aplicação)

O ângulo α varia entre 0 e 2π e vai ser dividido em *slices* e o ângulo β varia entre $-\pi/2$ e $\pi/2$ e vai ser dividido em *stacks*. Sabendo então que o que varia entre diferentes coordenadas esféricas são os ângulos anteriormente referidos podemos calcular todas as coordenadas para os pontos necessários.

$$x = raio * \cos(\beta) * \sin(\alpha)$$

$$y = raio * \sin(\beta)$$

$$z = raio * \cos(\beta) * \cos(\alpha)$$

A estratégia que escolhemos consiste então em desenhar dois triângulos a cada iteração.

4.2.5 Cilindro

Para a primitiva cilindro, tal como explicitado anteriormente, explicitamos o raio, a altura e as *slices* do mesmo. Começa-se por calcular o total de vértices dos triângulos que fazem parte do cilindro (7) bem como o ângulo de cada *slice* (8).

$$vertices_{total} = slices * 2 * 3$$

$$angulo = (2\pi)/slices$$

Recorremos a um ciclo for que, através das fórmulas trigonométricas, gera as diferentes coordenadas. A base do cilindro vai ser dividida em *slices* que irão dar origem a triângulos. Ao unir os vértices do topo e da base do cilindro conseguimos obter os triângulos que formam então as laterais do mesmo.

4.2.6 Toro

Para construir o Toro, procedemos à parametrização deste, segundo as fórmulas que se seguem e onde constam as variáveis R e r onde a primeira é o raio da origem ao interior do toro - “radius”, o segundo é o raio do anel interior do toro - “ring radius”, como se vê na figura seguinte.

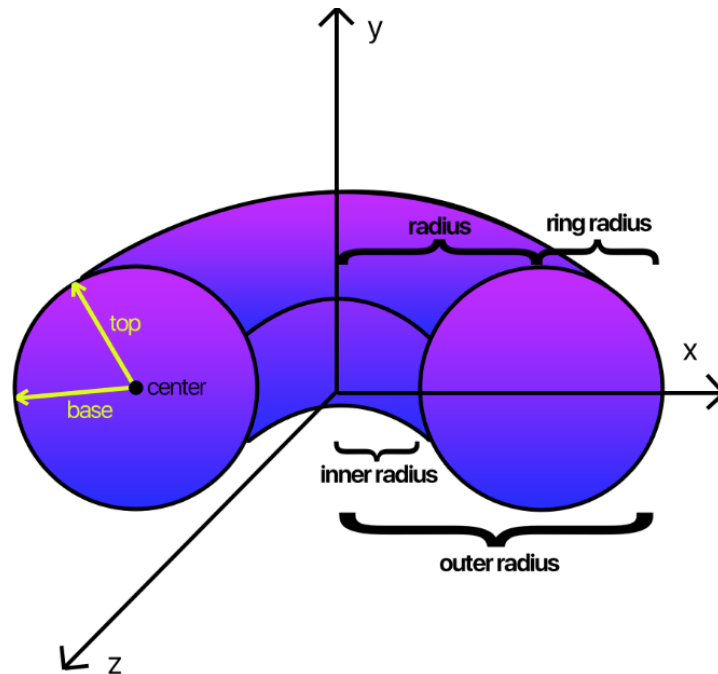


Figura 5: Variáveis representadas graficamente no Toro ¹

$$x = R * \cos\theta + r * \cos\phi * \cos\theta$$

$$y = R * \sin\theta + r * \cos\phi * \sin\theta$$

$$z = r * \sin\phi$$

$$0 \leq \phi < 2\pi$$

$$0 \leq \theta < 2\pi$$

A função responsável por esta criação designa-se por `createTorus` e recebe o conjunto de parâmetros seguintes: `in_radius`, `out_radius`, `slices`, `tacks`, `.3d filename`, que a partir dos dois primeiros conseguimos calcular o R e o r , necessários para gerar os pontos do Torus.

Listing 2: Parametrização do Toro

```
float R = ((out_radius - in_radius) / 2) + in_radius;
float r = ((out_radius - in_radius) / 2);
float teta = (M_PI * 2) / slices;
float fi = (M_PI * 2) / stacks;
Point p1, p2, p3, p4;
```

Do mesmo modo como se procedeu com a esfera, vamos dividir as variáveis ϕ e θ em stacks e slices, respetivamente. As variáveis `in_radius`, `out_radius` são o raio interior e o raio superior do toro respetivamente, bem visíveis na imagem.

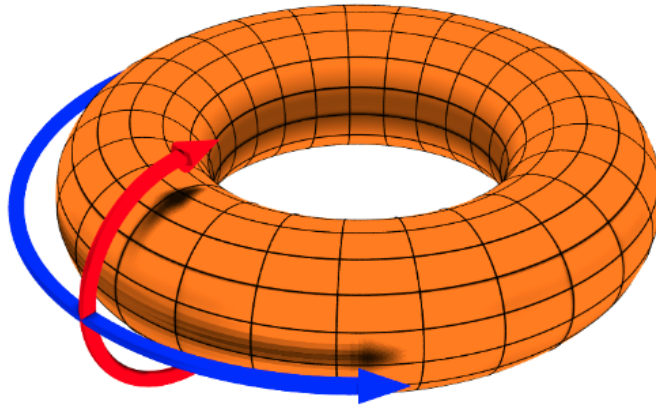


Figura 6: Toro dividido em *slices* e *stacks* ¹

1

¹Imagens retiradas dos apontamentos do Cesium acerca do trabalho de Computação Gráfica - <https://wide-joke-855.notion.site/Computa-o-Gr-fica-07c4e468437c41bd920deaef0985286a>

5 Motor

Sabendo que, para correr o *Engine*, é necessário um ficheiro .xml, decidiu-se usar a biblioteca do *tinyxml2.h*. Assim, conseguimos ler de uma só vez este ficheiro e alocar numa estrutura de dados denominada por **World**. Esta estrutura era constituída por outra estrutura (**Camera**) e por dois vetores: um que serve para alocar os ficheiros .3d e outro que guarda **Point**'s. Quanto à **Camera**, esta possui 4 arrays de *floats* referentes às posições e características da câmara. Em relação à estrutura **Point**, este possui 3 floats referentes às coordenadas de um ponto.

```
struct Camera {  
    float position[3] = { [0]: 0, [1]: 0, [2]: 0 };  
    float lookAt[3] = { [0]: 0, [1]: 0, [2]: 0 };  
    float up[3] = { [0]: 0, [1]: 0, [2]: 0 };  
    float projection[3] = { [0]: 0, [1]: 0, [2]: 0 };  
};  
  
struct World {  
    Camera *cam = new Camera;  
    vector<string> files;  
    vector<Point> points;  
};
```

Figura 7: Estruturas de dados **Camera** e **World**

Obtendo todos os ficheiros referidos no ficheiro .xml, tratou-se de fazer o *parse* dos pontos, guardando, como referido antes, no vetor pertencente à estrutura **World**. Seguido disto, faltava apenas alterar os detalhes da câmara: obter os ângulos α e β e a distância da câmara em relação à origem. Para isso, usou-se a posição inicial da câmara e fórmulas da trigonometria.

5.1 Extras

5.1.1 Movimentação da câmara

Quanto à câmara, esta tem a possibilidade de se movimentar graças à utilização da função *gluLookAt*. No entanto, a posição da câmara baseia-se no valor das variáveis globais α , β e *radius*.

$$radius = \sqrt{\text{pow}(\text{world.cam} \rightarrow \text{position}[0], 2) + \text{pow}(\text{world.cam} \rightarrow \text{position}[1], 2) + \text{pow}(\text{world.cam} \rightarrow \text{position}[2], 2)}$$

$$\beta = \arcsin(\text{world.cam} \rightarrow \text{position}[1] / radius)$$

$$\alpha = \arcsin(\text{world.cam} \rightarrow \text{position}[0] / \sqrt{\text{pow}(\text{world.cam} \rightarrow \text{position}[2], 2) + \text{pow}(\text{world.cam} \rightarrow \text{position}[0], 2)});$$

Nesse sentido, quando se recebe os parâmetros do ficheiro XML são atualizados esses três valores, de modo que a posição inicial da câmara seja a pretendida.

Para mover a câmara para cima ou para baixo da figura, basta clicar na tecla **UP** (aumentar beta) ou **DOWN** (diminuir beta), respetivamente, e para mover para a direita ou esquerda, basta clicar em **RIGHT** (aumentar alpha) ou **LEFT** (diminuir alpha), respetivamente. Esta funcionalidade é especificada pela função *processSpecialKeys*, que é posteriormente validada através da função *glutSpecialFunc* do GLUT.

Para aproximar ou afastar a câmara da origem, basta clicar em **I** ou **O**, respetivamente, aumentando e diminuindo *radius*. Esta funcionalidade é especificada pela função *processKeys*, que é posteriormente validada através da função *glutKeyboardFunc* do GLUT.

Além dos botões acima mencionados criamos também os botões **L** para alterar entre as linhas e a figura completa e o botão **A** para os eixos.

6 Testes

Para mostrar em funcionamento o trabalho exposto ao longo deste relatório realizamos os seguintes testes:

6.1 Cone

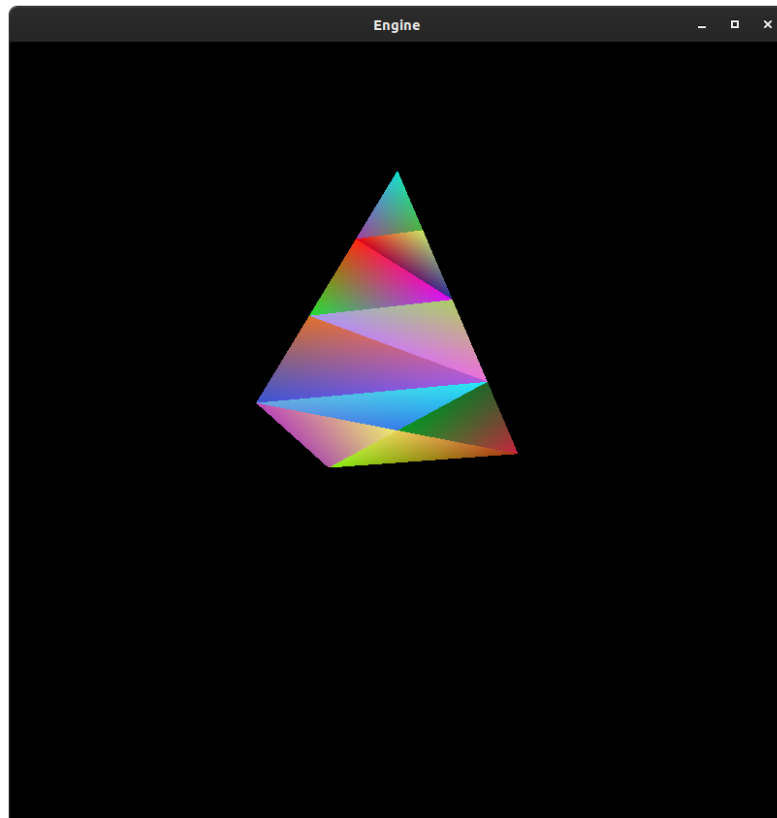


Figura 8: Comando: `generator cone 1 2 4 3 cone.3d`

6.2 Esfera

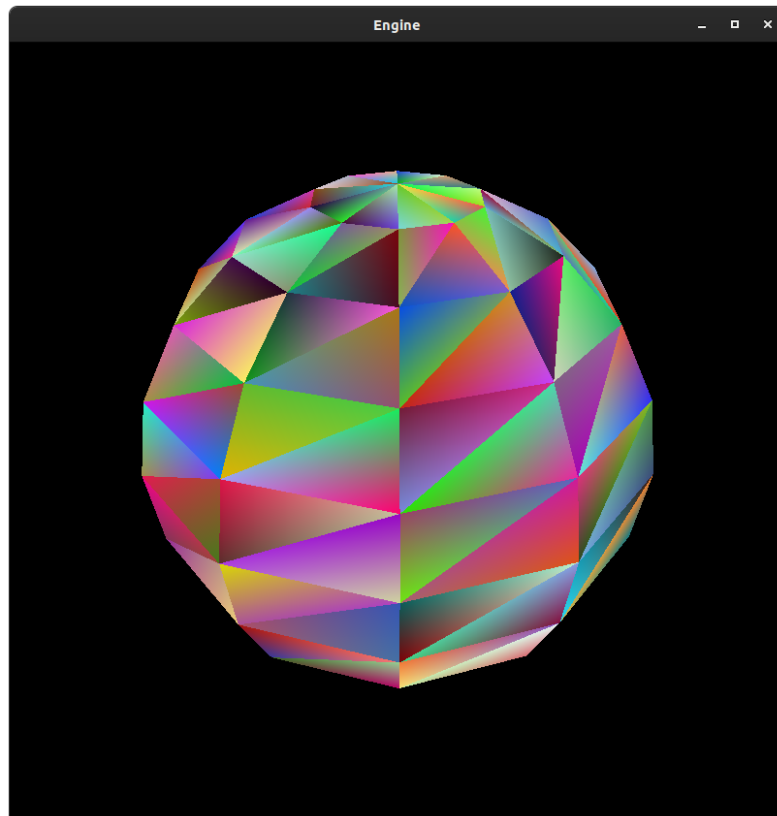


Figura 9: Comando: `generator sphere 1 10 10 sphere.3d`

6.3 Caixa

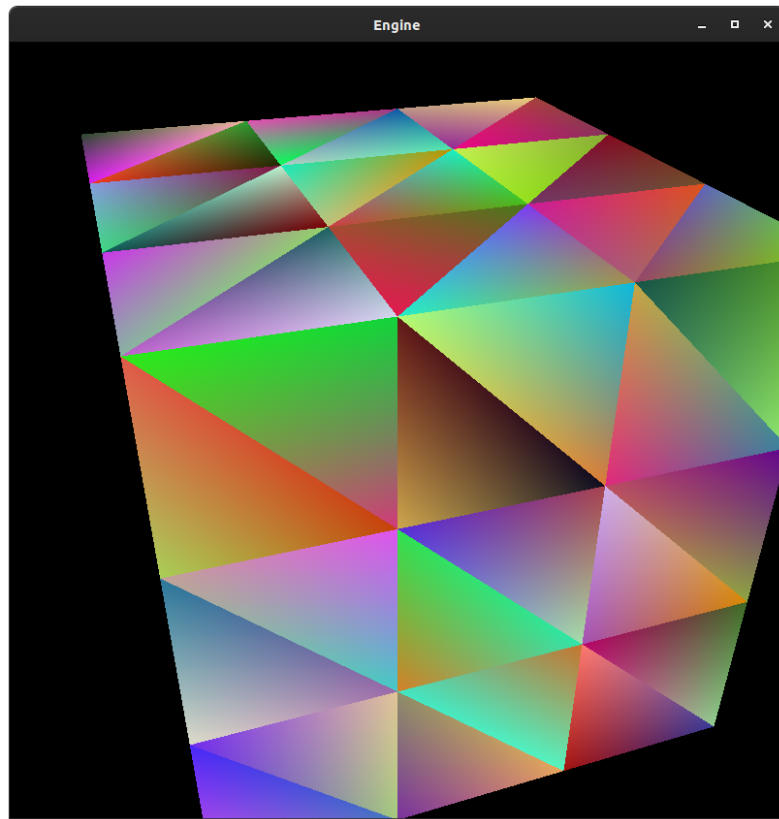


Figura 10: Comando: `generator box 2 3 box.3d`

6.4 Toro

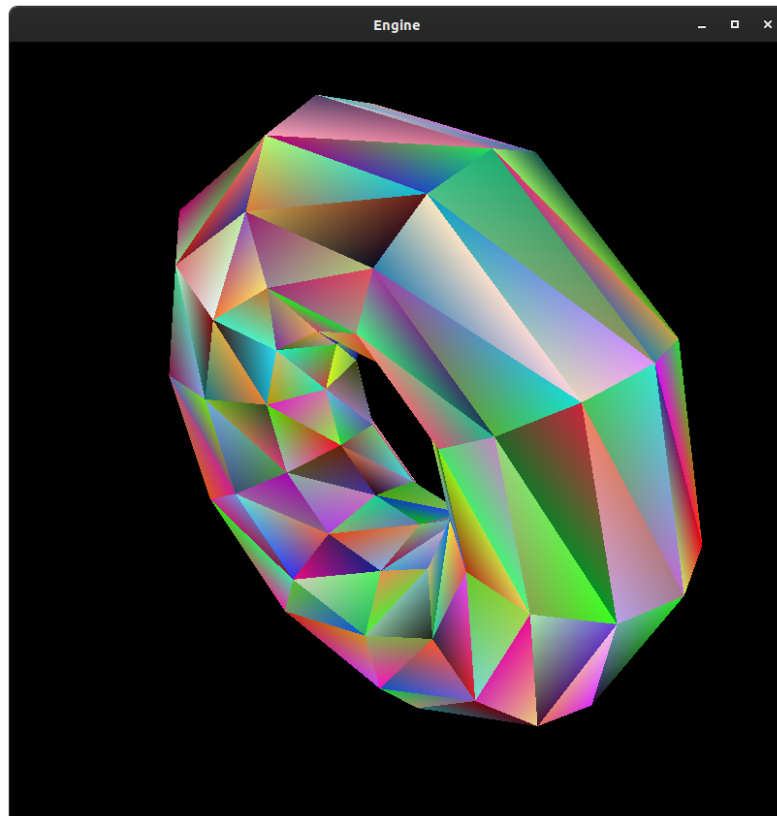


Figura 11: Comando: `generator torus 1 3 10 10 torus.3d`

6.5 Cilindro

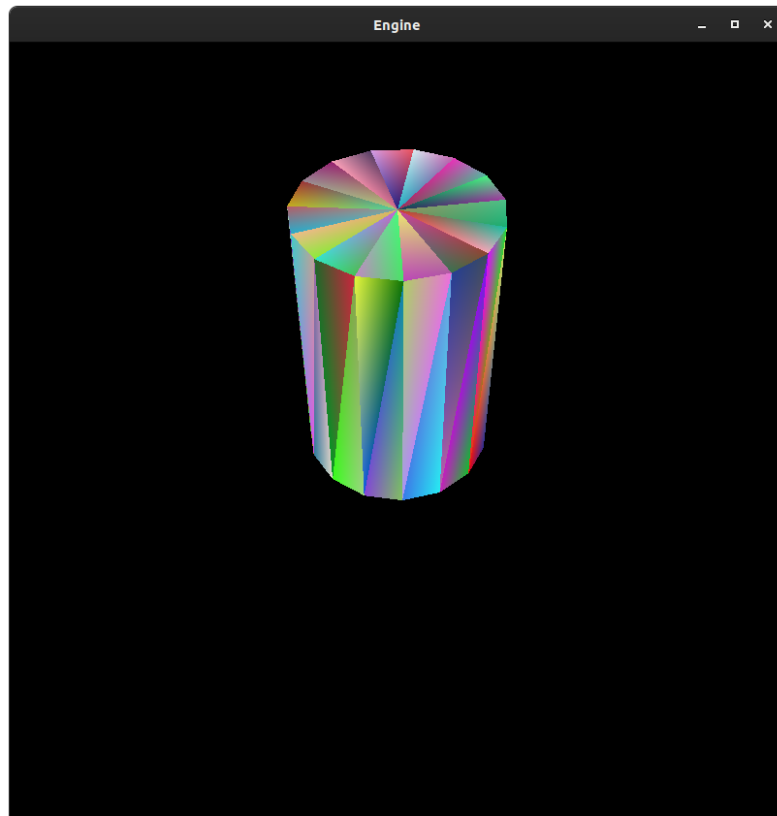


Figura 12: Comando: generator cylinder 1 3 15 cylinder.3d

6.6 Figuras sobrepostas

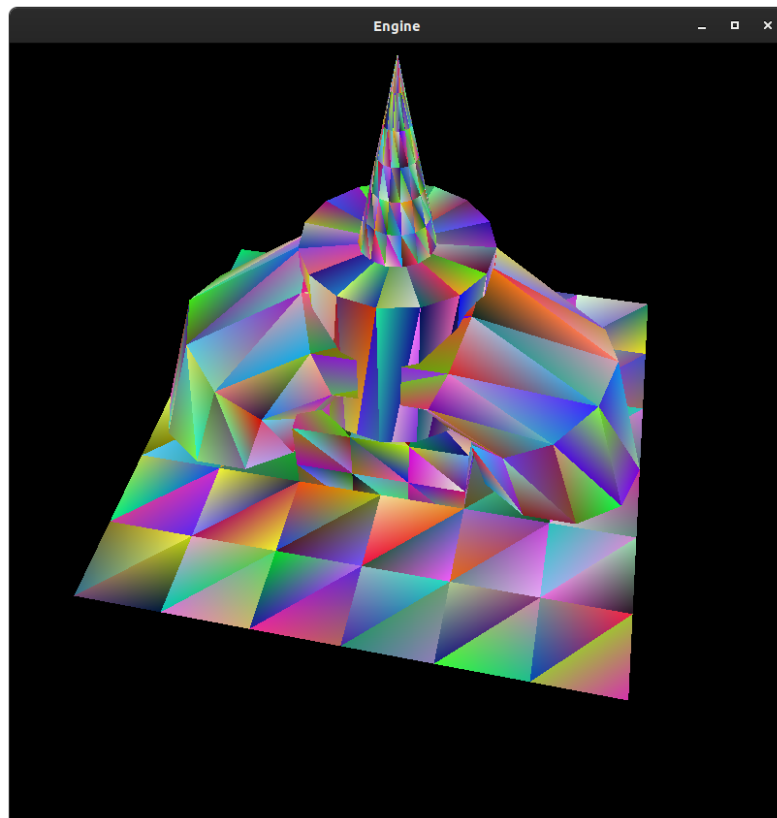


Figura 13: Sobreposição das figuras anteriores

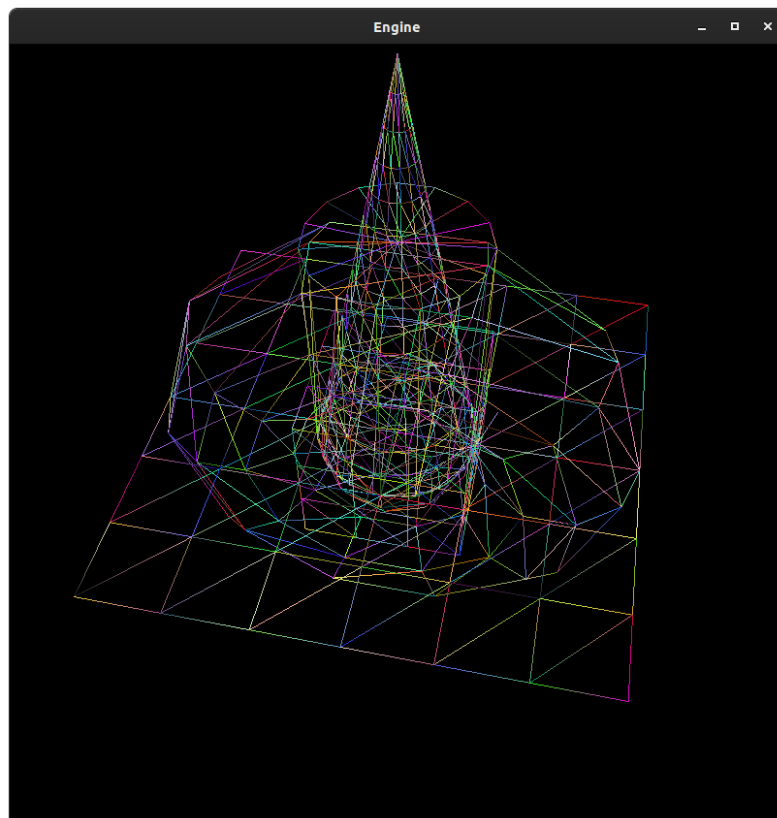


Figura 14: Comando: Sobreposição das figuras anteriores

7 Conclusão

Da realização desta fase inicial, o grupo considera que a mesma foi bem conseguida, já que se realizaram todas as funcionalidades requisitadas pelo próprio enunciado, acrescentando-se ainda as formas geométricas adicionais: o toro e cilindro.

Sumariamente, esta fase mostrou-se imprescindível para o desenvolvimento posterior do projeto, pois garantiu a compreensão do funcionamento geral do trabalho com modelos 3D e fomentou uma maior facilidade na manipulação destes com as bibliotecas utilizadas para o efeito. De um modo geral, contribuiu para uma maior familiarização com esta linguagem e “forma” de programar bem como construção de figuras que serão auxiliares nas fases seguintes.