



Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Computação Gráfica

Ano Letivo de 2022/2023

Graphical Primitives Phase 1

João Moreira a93326
Daniel Faria a93191
Paulo Filipe Cruz a80949

12 de março de 2023

CG

Índice

1	Introdução	1
2	Generator	2
2.1	Ficheiros 3D	2
2.2	Plano	3
2.3	Box	3
2.4	Cone	4
2.5	Esfera	5
2.6	Cilindro	6
3	Engine	7
4	Testes	9
4.1	Cone	9
4.2	Esfera	10
4.3	Caixa	11
4.4	Cilindro	12
4.5	Figuras sobrepostas	13
5	Conclusão	14

Lista de Figuras

2.1	Exemplo de um ficheiro .3d	2
2.2	Ciclo 'for' utilizado para a parte superior do cone	3
2.3	Ciclo 'for' utilizado para a parte superior do cone	4
2.4	Ciclo 'for' utilizado para a parte inferior do cone	4
2.5	Ciclo 'for' utilizado para a criacao dos pontos da esfera	5
2.6	Ciclo 'for' utilizado para a criacao dos pontos do cilindro	6
3.1	Estruturas de dados CameraParameters , Group e WorldInfo	7

4.1	Comando: generator cone 1 2 4 3 cone_1_2_4_3.3d	9
4.2	Comando: generator sphere 1 10 10 sphere_1_10_10.3d	10
4.3	Comando: generator box 2 3 box_2_3.3d	11
4.4	Comando: generator cylinder 1 3 15 cylinder_1_3_15.3d	12
4.5	Esfera e Plano	13

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.

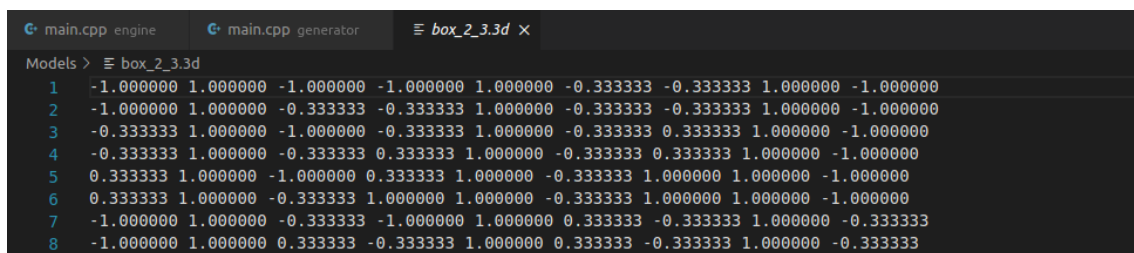
Naturalmente, é indispensável que o conjunto de objetivos supracitados seja concretizado com sucesso e, para isso, o formato do relatório está organizado de acordo com uma descrição do problema inicial, seguindo-se o conjunto de aspetos relevantes em relação ao *Generator* e chegando aos aspetos primordiais sobre o *Engine*.

2 Generator

O *generator* cria os ficheiros .3d com as primitivas para serem usados pelo *engine*.

2.1 Ficheiros 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. Esta formatação respeita a regra da mão direita.



```
main.cpp engine  main.cpp generator  box_2_3.3d X
Models > box_2_3.3d
1  -1.000000 1.000000 -1.000000 -1.000000 1.000000 -0.333333 -0.333333 1.000000 -1.000000
2  -1.000000 1.000000 -0.333333 -0.333333 1.000000 -0.333333 -0.333333 1.000000 -1.000000
3  -0.333333 1.000000 -1.000000 -0.333333 1.000000 -0.333333 0.333333 1.000000 -1.000000
4  -0.333333 1.000000 -0.333333 0.333333 1.000000 -0.333333 0.333333 1.000000 -1.000000
5  0.333333 1.000000 -1.000000 0.333333 1.000000 -0.333333 1.000000 1.000000 -1.000000
6  0.333333 1.000000 -0.333333 1.000000 1.000000 -0.333333 1.000000 1.000000 -1.000000
7  -1.000000 1.000000 -0.333333 -1.000000 1.000000 0.333333 -0.333333 1.000000 -0.333333
8  -1.000000 1.000000 0.333333 -0.333333 1.000000 0.333333 -0.333333 1.000000 -0.333333
```

Figura 2.1: Exemplo de um ficheiro .3d

2.2 Plano

O *Plano* é gerado tendo como base um ciclo que cria dois triângulos, que compartilham dois pontos para formar um quadrado. A junção de todos esses quadrados formam um plano. Para garantir que os triângulos estejam voltados para cima, a regra da mão direita é usada, o que significa que os pontos devem ser desenhados no sentido contrário dos ponteiros do relógio para que tal seja verificado. As dimensões do plano podem ser variáveis, sendo que um valor de comprimento é dividido por 2 de modo a que o plano esteja centrado na origem do referencial.

```
float slice = length/divisions;
std::string filename = args[4];
Point vertices[2][2];

ofstream planeFile(filename, ios::out | ios::app | ios::binary);

for(i=-divisions/2; i*slice<length/2; i++){
    for(j=-divisions/2; j*slice<length/2; j++){

        //P1.
        vertices[0][0].x = j*slice;
        vertices[0][0].y = length/2;
        vertices[0][0].z = i*slice;

        //P2.
        vertices[0][1].x = (j+1)*slice;
        vertices[0][1].y = length/2;
        vertices[0][1].z = i*slice;

        //P3.
        vertices[1][0].x = (j+1)*slice;
        vertices[1][0].y = length/2;
        vertices[1][0].z = (i+1)*slice;

        //P4.
        vertices[1][1].x = j*slice;
        vertices[1][1].y = length/2;
        vertices[1][1].z = (i+1)*slice;

        //
        P1.
        planeFile << toString(vertices[0][0]) << " " << toString(vertices[1][1]) << " " << toString(vertices[0][1]) << endl;
        //
        P4.
        P3.
        P2.
        planeFile << toString(vertices[1][1]) << " " << toString(vertices[1][0]) << " " << toString(vertices[0][1]) << endl;
    }
}

planeFile.close();
```

Figura 2.2: Ciclo 'for' utilizado para a parte superior do cone

2.3 Box

Na criação da *Box* são utilizados 6 pares de ciclos 'for', cada um faz a criação de um plano tendo em consideração o lado da caixa que representa. Cada um desses pares de ciclos 'for' tem uma lógica análoga aos ciclos usados na função responsável pela criação do *Plano*.

2.4 Cone

A criação do cone foi feita considerando o mesmo em duas partes, o topo (a última "peça do cone") e a base. Assim sendo, utilizamos 1 ciclo for e 1 par de ciclos for. O primeiro é responsável pela criação dos pontos da parte superior, sendo que itera apenas nas slices, e o último pelos pontos da base. Os pontos são calculados como coordenadas polares.

```
//Top
for (i=0; i<slices; i++) {
    //P1
    vertices[0][0].x = 0;
    vertices[0][0].y = 0;
    vertices[0][0].z = 0;

    //P2
    vertices[0][1].x = radius * sin((i + 1) * angulo);
    vertices[0][1].y = 0;
    vertices[0][1].z = radius * cos((i + 1) * angulo);

    //P3
    vertices[1][0].x = radius * sin(i * angulo);
    vertices[1][0].y = 0;
    vertices[1][0].z = radius * cos(i * angulo);

    //
    coneFile << toString(vertices[0][0]) + " " + toString(vertices[0][1]) + " " + toString(vertices[1][0]) << endl;
}
```

Figura 2.3: Ciclo 'for' utilizado para a parte superior do cone

```
//Bottom
for (i = 0; i < slices; i++) {
    for (j = 0; j < stacks; j++) {
        //
        vertices[0][0].x = (radius - (j * (radius / stacks))) * sin(i * angulo);
        vertices[0][0].y = j * (height / stacks);
        vertices[0][0].z = (radius - (j * (radius / stacks))) * cos(i * angulo);

        vertices[0][1].x = (radius - (j * (radius / stacks))) * radius * sin((i+1) * angulo);
        vertices[0][1].y = j * (height / stacks);
        vertices[0][1].z = (radius - (j * (radius / stacks))) * cos((i+1) * angulo);

        vertices[1][0].x = (radius - ((j+1) * (radius / stacks))) * radius * sin((i+1) * angulo);
        vertices[1][0].y = (j+1) * (height / stacks);
        vertices[1][0].z = (radius - ((j+1) * (radius / stacks))) * cos((i+1) * angulo);

        vertices[1][1].x = (radius - ((j+1) * (radius / stacks))) * radius * sin(i * angulo);
        vertices[1][1].y = (j+1) * (height / stacks);
        vertices[1][1].z = (radius - ((j+1) * (radius / stacks))) * cos(i * angulo);

        //
        coneFile << toString(vertices[0][0]) + " " + toString(vertices[0][1]) + " " + toString(vertices[1][0]) << endl;
        //
        coneFile << toString(vertices[1][0]) + " " + toString(vertices[1][1]) + " " + toString(vertices[0][0]) << endl;
    }
}
coneFile.close();
```

Figura 2.4: Ciclo 'for' utilizado para a parte inferior do cone

2.5 Esfera

A construção da esfera é realizada usando duas dimensões de ciclos. O primeiro ciclo itera sobre as divisões horizontais da esfera, enquanto o segundo ciclo itera sobre as divisões verticais da esfera. As coordenadas esféricas dos seus pontos foram calculadas com o recurso aos ângulos α e β que variam dependendo da iteração do ciclo em que nos encontramos. α corresponde a $2\pi/\text{slices}$ e β corresponde a π/stacks . Para garantir a centralização da esfera, o ciclo de stacks varia de $-\text{stacks}/2$ a $\text{stacks}/2$. No final de cada iteração do ciclo os triângulos são escritos no respectivo ficheiro (`"sphereFile"`).

```
for (int j = -stacks/2; j < stacks/2; j++) {
    beta=(j*bStep);
    for (int i = 0; i < slices; i++) {
        alpha=(aStep*i);

        //P1
        vertices[0][0].x = radius*cosf(beta)*sinf(alpha);
        vertices[0][0].y = radius*sinf(beta);
        vertices[0][0].z = radius*cosf(beta)*cosf(alpha);

        //P2
        vertices[0][1].x = radius*cosf(beta)*sinf(aStep*(i+1));
        vertices[0][1].y = vertices[0][0].y;
        vertices[0][1].z = radius*cosf(beta)*cosf(aStep*(i+1));

        //P3
        vertices[1][0].x = radius*cosf((j+1)*bStep)*sinf(aStep*(i+1));
        vertices[1][0].y = radius*sinf((j+1)*bStep);
        vertices[1][0].z = radius*cosf((j+1)*bStep)*cosf(aStep*(i+1));

        //P4
        vertices[1][1].x = radius*cosf((j+1)*bStep)*sinf(alpha);
        vertices[1][1].y = vertices[1][0].y;
        vertices[1][1].z = radius*cosf((j+1)*bStep)*cosf(alpha);

        //
        sphereFile << toString(vertices[0][0]) + " " + toString(vertices[0][1]) + " " + toString(vertices[1][0]) << std::endl;
        //
        sphereFile << toString(vertices[1][0]) + " " + toString(vertices[1][1]) + " " + toString(vertices[0][0]) << std::endl;
    }
}
sphereFile.close();
std :: cout << "Generator: " + filename + " written successfully!\n";
}
```

Figura 2.5: Ciclo 'for' utilizado para a criação dos pontos da esfera

2.6 Cilindro

Para a figura do cilindro, 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 bem como o ângulo de cada *slice*.

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.

```
float sectorStep = M_PI * 2 / slices;

for (int i = 0; i <= slices; i++) {
    float alpha = i * sectorStep;
    float nextAlpha = sectorStep + alpha;

    p1.x = radius * sin(alpha); p1.y = height; p1.z = radius * cos(alpha);
    p2.x = radius * sin(nextAlpha); p2.y = height; p2.z = radius * cos(nextAlpha);

    p3.x = 0.0; p3.y = height; p3.z = 0.0;

    // base top
    cylinderFile << toString(p3) + " " + toString(p1) + " " + toString(p2) << std::endl;

    p3.y = 0.0;
    p2.y = 0.0;
    p1.y = 0.0;
    // base bot
    cylinderFile << toString(p3) + " " + toString(p1) + " " + toString(p2) << std::endl;

    p3.x = p1.x; p3.z = p1.z;
    p2.y = height;
    p1.y = height;
    // side1
    cylinderFile << toString(p3) + " " + toString(p2) + " " + toString(p1) << std::endl;

    p1.x = p2.x; p1.y = 0.0; p1.z = p2.z;
    // side2
    cylinderFile << toString(p3) + " " + toString(p1) + " " + toString(p2) << std::endl;
}

cylinderFile.close();
```

Figura 2.6: Ciclo 'for' utilizado para a criação dos pontos do cilindro

3 Engine

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 pela *width* e *height* da janela, uma estrutura (**CameraParameters**) e por um vetor **Group** que, por sua vez, possui um vetor que guarda **Triangle**'s. Quanto à **CameraParameters**, esta possui 3 estruturas do tipo **Point**, sendo estas, *position*, *lookAt* e *up* e 3 *floats*, nomeadamente, *fov*, *near* e *far*, referentes às posições e características da câmara. Em relação à estrutura **Point**, este possui 3 *floats* referentes às coordenadas de um ponto. Por fim, a estrutura **Triangle** é constituído por 3 **Point**'s.

```
26 struct CameraParameters {
27     Point position;
28     Point lookAt;
29     Point up;
30     float fov;
31     float near;
32     float far;
33 };
34
35 struct Group{
36     std :: vector<Triangle> triangles;
37 };
38
39 struct WorldInfo {
40     int width;
41     int height;
42     CameraParameters camera;
43     std :: vector<Group> groups;
44     //std :: vector<std :: vector<Triangle>> points;
45     //std::vector<std::string> models;
46     // std::vector<Triangle> world_triangles;
47 };
48
49
50 WorldInfo world;
```

Figura 3.1: Estruturas de dados **CameraParameters**, **Group** e **WorldInfo**

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 **WorldInfo**. Seguido disto,

faltava apenas alterar os detalhes da câmara. Com todos os detalhes guardados, procedeu-se à representação dos dados, a partir do *OpenGL*.

4 Testes

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

4.1 Cone

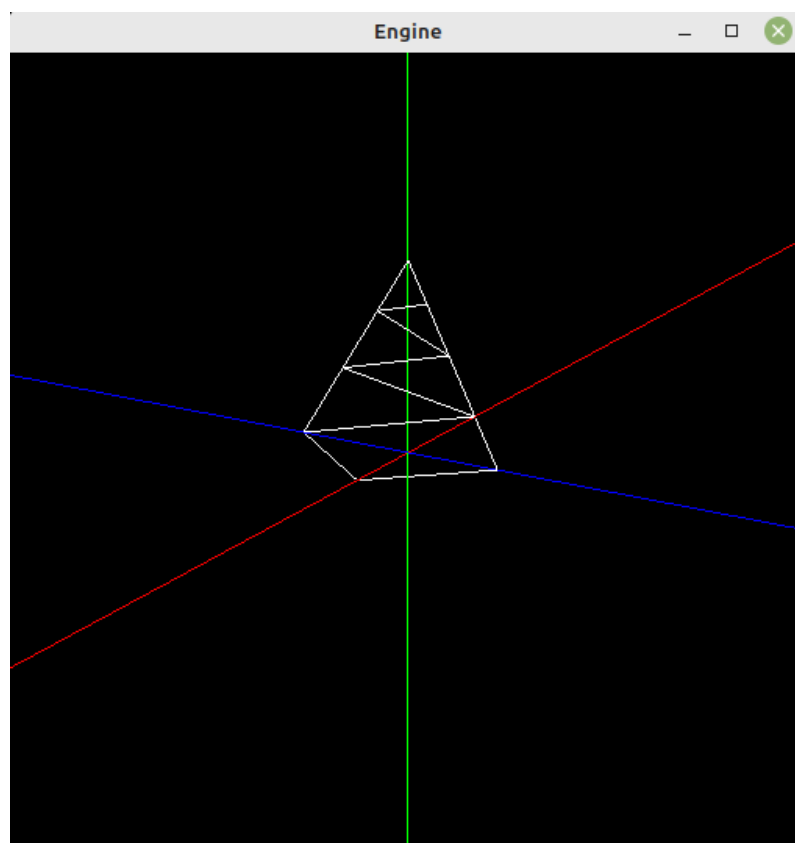


Figura 4.1: Comando: `generator cone 1 2 4 3 cone_1_2_4_3.3d`

4.2 Esfera

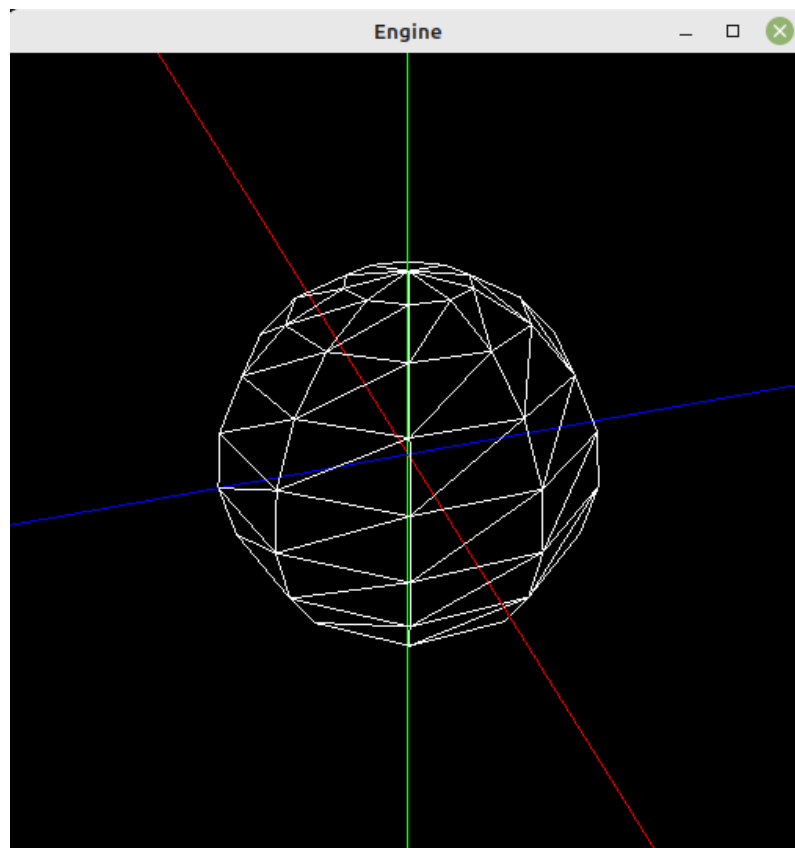


Figura 4.2: Comando: `generator sphere 1 10 10 sphere_1_10_10.3d`

4.3 Caixa

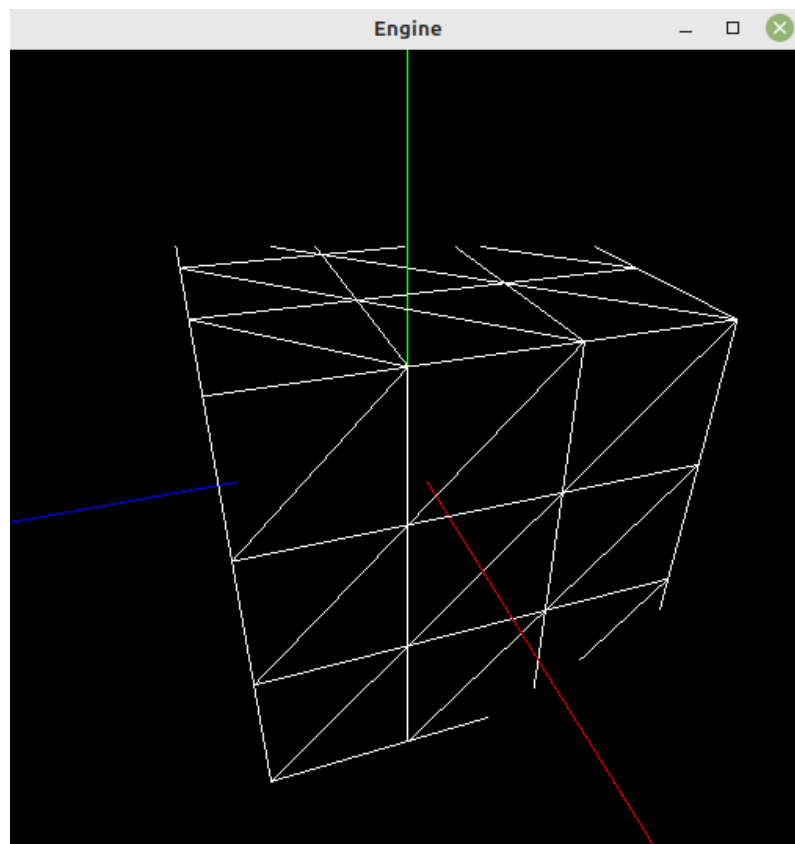


Figura 4.3: Comando: `generator box 2 3 box_2_3.3d`

4.4 Cilindro

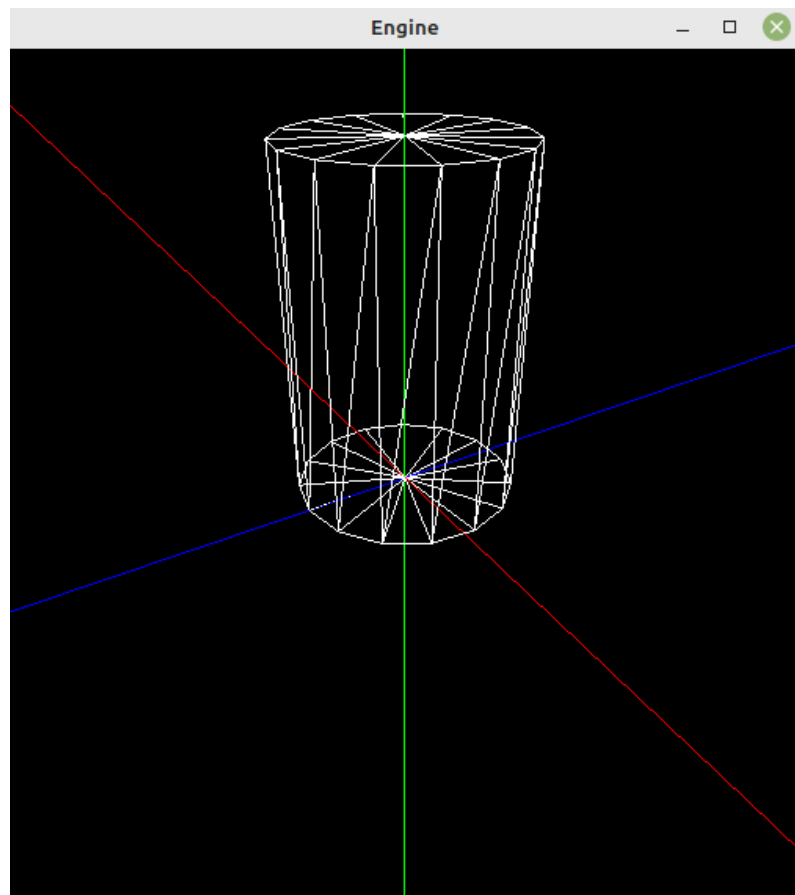


Figura 4.4: Comando: `generator cylinder 1 3 15 cylinder_1_3_15.3d`

4.5 Figuras sobrepostas

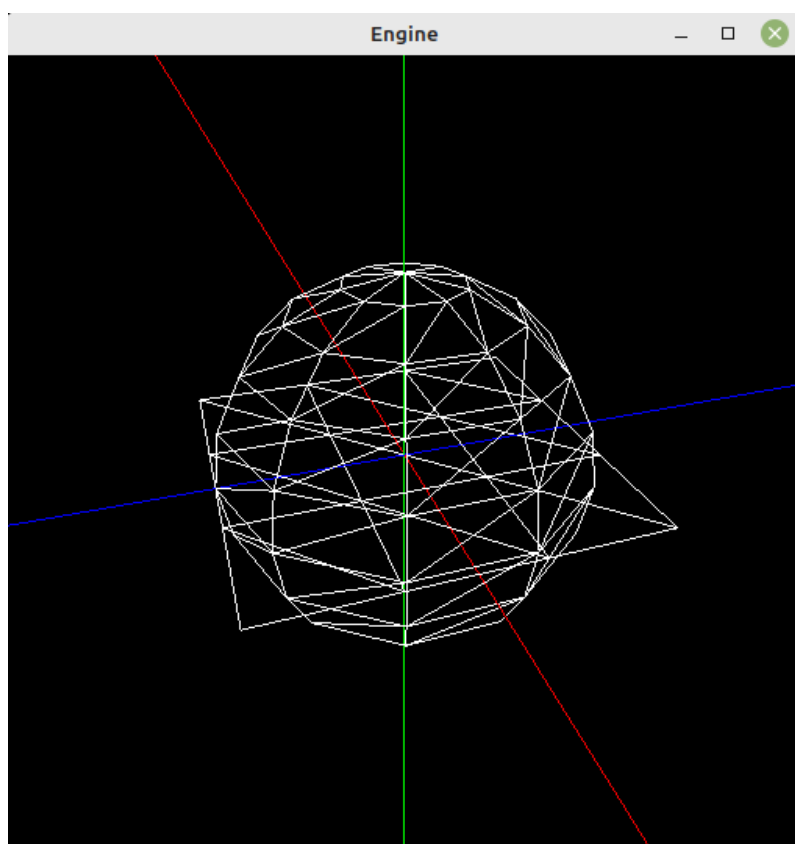


Figura 4.5: Esfera e Plano

5 Conclusão

O grupo considera que a fase inicial foi bem-sucedida, pois todas as funcionalidades solicitadas no enunciado foram realizadas, além de formas geométricas adicionais, como o cilindro. Essa fase foi fundamental para o desenvolvimento subsequente do projeto, pois permitiu entender o funcionamento geral dos modelos 3D e facilitou a manipulação desses modelos com as bibliotecas utilizadas.

Em geral, essa fase contribuiu para uma maior familiarização com a linguagem de programação e a construção de figuras que serão úteis nas próximas fases do projeto.