

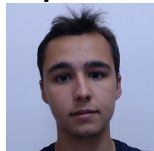
Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Unidade Curricular de Computação Gráfica

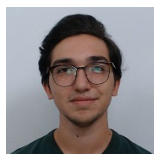
Ano Letivo de 2022/2023

Primitivas Gráficas Fase 1

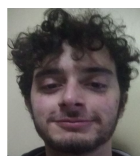
Grupo 15



a84696, Renato Gomes



a89486, Tomás Dias



a97223, João Castro

URL do Repositório
<https://github.com/joaocast/CG2022-23>

March 7, 2023

Índice

1	Introdução	1
2	Arquitetura do Sistema	2
3	Implementação	3
3.1	Generator	3
3.1.1	Primitivas	3
3.2	Engine	7
4	Manual de Utilização	8
5	Testes	9
6	Conclusão	14

Lista de Figuras

2.1	Arquitetura do sistema	2
4.1	Ficheiro de configuração XML por default	8
5.1	Resultado de test_1_1.xml	9
5.2	Resultado de test_1_2.xml	10
5.3	Resultado de test_1_3.xml	10
5.4	Resultado de test_1_4.xml	11
5.5	Resultado de test_1_5.xml	12
5.6	Prisma	13
5.7	Cilindro	13

1. Introdução

Este relatório apresenta a primeira fase do nosso projeto, que envolve duas aplicações. A primeira aplicação é um gerador de ficheiros com informações de modelos e a segunda aplicação é o motor que lê um arquivo de configuração escrito em XML e exibe os modelos.

Para criar os arquivos de modelo, o ***generator*** receberá parâmetros como o tipo de primitiva gráfica, outros parâmetros necessários para a criação do modelo e o arquivo de destino onde as informações serão armazenados.

A segunda aplicação, o ***engine***, receberá um arquivo de configuração escrito em XML que conterá as configurações da câmera e a indicação de quais arquivos de modelos gerados pelo gerador devem ser carregados.

A metodologia aplicada ao longo desta fase foi reuniões frequentes com o grupo todo presente, dividindo tarefas de semana para semana.

2. Arquitetura do Sistema

O presente sistema encontra-se organizado em dois módulos, a engine e o generator. O módulo da engine possui um ficheiro engine.cpp e engine.h onde se encontram declarados os métodos e variáveis usadas no .cpp correspondente. Para além destes, estão presentes os ficheiros tinyxml2.cpp e tinyxml2.h, necessários para realizar a leitura e parsing dos ficheiros de configuração em formato XML. Por outro lado, foram adicionados dois folders, um com capturas de ecrã dos resultados dos testes realizados e outro com as capturas de ecrã das primitivas expectáveis.

O módulo generator possui o ficheiro generator.cpp e o generator.h. Consequentemente, após a geração das primitivas especificadas pelo utilizador, será criada uma pasta adicional, "models", com todos os ficheiros 3d, onde irão estar descritos todos os pontos do modelo correspondente.



```
joao@ubuntu:~/CG2022-23/FASE1$ tree -I 'ALL*|class*'
.
├── engine_module
│   ├── build
│   │   └── Release
│   │       ├── box_2_3_3d
│   │       ├── box_3d
│   │       ├── cone_1_2_4_3_3d
│   │       ├── cone_3d
│   │       ├── config.xml
│   │       ├── cylinder_3d
│   │       ├── engine.exe
│   │       ├── plane_2_3_3d
│   │       ├── plane_3d
│   │       ├── sphere_1_10_10_3d
│   │       ├── sphere_3d
│   │       ├── test_1_1.xml
│   │       ├── test_1_2.xml
│   │       ├── test_1_3.xml
│   │       ├── test_1_4.xml
│   │       └── test_1_5.xml
│   ├── engine
│   │   ├── engine.cpp
│   │   ├── engine.h
│   │   ├── test_files_phase1
│   │   │   ├── test_1_1.png
│   │   │   ├── test_1_2.png
│   │   │   ├── test_1_3.png
│   │   │   ├── test_1_4.png
│   │   │   └── test_1_5.png
│   │   ├── test_results_phase1
│   │   │   ├── test1.JPG
│   │   │   ├── test2.JPG
│   │   │   ├── test3.JPG
│   │   │   ├── test4.JPG
│   │   │   └── test5.JPG
│   │   ├── tinyxml2.cpp
│   │   └── tinyxml2.h
│   └── generator_module
│       ├── build
│       │   ├── generator.sln
│       │   └── Release
│       │       └── generator.exe
│       ├── generator
│       │   ├── generator.cpp
│       │   └── generator.h
│       └── models
├── engine_module
├── generator_module
├── models
├── test_files_phase1
├── test_results_phase1
├── tinyxml2.cpp
├── tinyxml2.h
└── generator_module
```

10 directories, 34 files
joao@ubuntu:~/CG2022-23/FASE1\$

Figura 2.1: Arquitetura do sistema

3. Implementação

3.1 Generator

3.1.1 Primitivas

Plano

Para a construção do cubo procedeu-se inicialmente ao quociente entre o comprimento do lado do cubo (units) e o número de grades/grelhas (grid), que irão formar $grid^2$ quadrados. O valor desta divisão irá representar o *step* quando for necessário construir cada um destes triângulos. Os triângulos representados serão triângulos retângulos com catetos de igual comprimento, pelo que será possível obtermos um quadrado com 2 triângulos deste tipo. Logo, o valor do *step* será o valor de um desses lados. A partir desta informação e tendo noção que o *Y* é sempre igual a 0 para todos os pontos conseguimos então gerar todos os pontos necessários para criar os vértices do plano,

Para tal vamos utilizar dois ciclos for aninhados em que vamos calcular os pontos da seguinte forma:

$$\begin{aligned} px1 &= -finalx + (j * stepx); \\ px2 &= -finalx + ((j + 1) * stepx); \\ pz1 &= -finalz + (i * stepz); \\ pz2 &= -finalz + ((i + 1) * stepz); \end{aligned}$$

sendo que:

$$\begin{aligned} -finalz + i * stepz &< finalz; \\ -finalx + j * stepx &< finalx; \\ \text{para } i, j \in N, \text{ } stepx &= stepz = units/grid \end{aligned}$$

Cubo

Para a construção do cubo procedeu-se inicialmente ao quociente entre o comprimento do lado do cubo (units) e o número de grades/grelhas (grid), que irão formar $grid^2$ quadrados para cada uma das faces. O valor desta divisão irá representar o 'step' quando for necessário construir cada um destes triângulos. Os triângulos representados serão triângulos retângulos com catetos de igual comprimento, pelo que será possível obtermos um quadrado com 2 triângulos deste tipo. Logo, o valor do step será o valor de um desses lados. De seguida, procedeu-se à divisão por 2 do comprimento do lado do cubo (units), de forma a pudermos iterar dentro deste intervalo $[-units/2, units/2]$, uma vez que o cubo tem de estar obrigatoriamente centrado na origem.

De seguida, procedeu-se à construção de planos paralelos, ou seja, em que apenas um dos componentes é sempre constante. Se considerarmos como exemplo a face inferior e a face superior do cubo (são paralelas entre si), todos os pontos de baixo terão um $y < 0$ constante e os pontos de cima terão um $y > 0$ (simétrico) constante. Logo, a partir dos valores do step e do intervalo mencionado em cima, vamos realizar ciclos for, de modo a irmos construindo cada um dos pontos dos triângulos de ambas as faces. Assim, um quadrado constituído por dois triângulos pertencentes ao plano perpendicular ao semi-eixo negativo Oy irão apresentar os seguintes pontos:

$p1(-finalz + ((i + 1) * stepz), -finaly, -finalz + ((i + 1) * stepz));$
 $p2(-finalx + (j * stepx), -finaly, -finalz + ((i + 1) * stepz));$
 $p3(-finalx + (j * stepx), -finaly, -finalz + (i * stepz))$
 $p4(-finalx + ((j + 1) * stepx), -finaly, -finalz + (i * stepz))$

Sendo que:

$-finalz + i * stepz < finalz$ e $-finalx + j * stepx < finalx$, para $i, j \in N$,
 $stepx = stepz = units/grid$ e $finalz = finaly = units/2$

Os triângulos do plano superior terão da mesma forma apenas a componente y simétrica. A construção das restantes faces do cubo seguem o mesmo raciocínio.

Esfera

Inicialmente, são definidos os valores de deltaAlpha e deltaBeta, que serão usados para gerar todas os pontos da esfera.

$deltaAlpha = 2 * \pi / slices$
 $deltaBeta = \pi / stacks$

O algoritmo, então, começa a criar os triângulos que formarão a esfera, percorrendo todas as stacks e slices necessárias. Para cada stack, é alterado o valor de beta, enquanto que para cada slice, é alterado o valor de alpha. Isto é:

```

beta = i * deltaBeta;
nextBeta = (i + 1) * deltaBeta
alpha = j * deltaAlpha
nextAlpha = (j + 1) * deltaAlpha

```

Sendo que:

$-(stacks/2) < i < (stacks/2)$ e $0 < j < slices$, para $i, j \in N$

Para cada ponto da esfera, são calculadas as coordenadas x, y e z utilizando as fórmulas matemáticas apropriadas ($radius * \cos(beta) * \sin(alpha)$, $radius * \sin(beta)$, $radius * \cos(beta) * \cos(alpha)$) com base no raio, beta e alpha. Em seguida, quatro pontos são criados a partir dos quais dois triângulos.

Por fim, a partir dos dois triângulos, que são armazenados em uma *vector* de triângulos, é formado um quadrado. Após todos os pontos de todos os triângulos serem calculados escreve-se para o ficheiro .3d o numero de pontos e de seguida os pontos (Um por linha).

Cone

Inicialmente, cria-se a base do cone, que é um círculo formado por slices triangulares. O raio do círculo é definido pelo parâmetro *radius*, e o ângulo entre os triângulos é calculado pela fórmula $(M_PI * 2) / slices$ (*stepAlpha*).

Depois, cria-se as fatias horizontais do cone, começando pela base e subindo até à penúltima *stack* do cone. Para cada *stack*, recalcula-se o raio da stack, que vai diminuindo à medida que o cone é construído.

Para isto as formulas utilizadas foram:

```

Point 1 = (curRad * sin(alpha), curHeight, curRad * cos(alpha)),
Point 2 = (curRad * sin(alpha + step), curHeight, curRad * cos(alpha + step)),
Point 3 = ((curRad-radStep)*sin(alpha+step), curHeight+stackStep, (curRad-radStep) * cos(alpha + step)),
Point 4 = ((curRad - radStep) * sin(alpha), curHeight + stackStep, (curRad - radStep) * cos(alpha))

```

Sendo que:

```

curHeight = AlturaAtual & curRad = RaioAtual,
stackStep = height ÷ stacks & radStep = radius ÷ stacks

```

Por fim, cria-se a parte superior do cone, que se trata de triângulos que ligam os pontos da última fatia horizontal à ponta do cone.

Cilindro & Prisma

Inicialmente, começa-se por criar os triângulos que formam as bases inferior e superior do cilindro. Para isso, percorre-se as fatias do cilindro, criando triângulos com vértices na borda da fatia e no centro da base, de acordo com o raio específico (b_rad e t_rad), dependendo se esses raios são iguais ou não, teremos um cilindro ou um prisma.

Em seguida, inicia-se a construção da superfície lateral do cilindro. Percorre-se assim, novamente, as fatias do cilindro, mas desta vez para cada fatia ela cria um conjunto de triângulos que ligam as camadas horizontais do cilindro. O raio e a altura dos vértices são calculados para cada camada e fatia, sendo que no caso de um cilindro esses raios são sempre iguais.

Para isto as formulas utilizadas foram:

Point 1 = ($curRad * \sin(\alpha)$, $curHeight$, $curRad * \cos(\alpha)$)
Point 2 = ($curRad * \sin(\alpha + step)$, $curHeight$, $curRad * \cos(\alpha + step)$)
Point 3 = ($(curRad - radStep) * \sin(\alpha + step)$, $curHeight - stackStep$, $(curRad - radStep) * \cos(\alpha + step)$)
Point 4 = ($(curRad - radStep) * \sin(\alpha)$, $curHeight - stackStep$, $(curRad - radStep) * \cos(\alpha)$)

Sendo que:

$step = (\pi * 2) \div slices$
 $stackStep = height \div stacks$
 $radStep = \text{abs}(b_rad - t_rad) \div stacks$
 $h_height = height \div 2$

3.2 Engine

Ao executar a engine, inicialmente, é realizado o parse do ficheiro XML. A nossa equipa de trabalho optou por passar como argumento na execução da função main, o nome do ficheiro XML. Caso contrário, por default será considerado o ficheiro "config.xml". Neste ficheiro estarão algumas configurações da câmara, definições do display da janela e quais os modelos que devem ser apresentados.

A partir do tinyxml2 foi possível armazenar em memória as configurações acima mencionadas. Para isso em primeiro lugar, foi declarado um apontador XMLDocument, e de seguida foi invocado o LoadFile com o path do ficheiro como argumento. A abertura do ficheiro será bem sucedida caso retorne 0. Posteriormente, foi possível fazer uma análise dos campos desejados recorrendo a apontadores XMLElement*. Quando uma tag A está aninhada numa tag B, realizou-se um A->FirstChildElement, de modo a obtermos um apontador XMLElement* da tag B. Caso as tags A e B estejam ao mesmo nível realizou-se um A->NextSiblingElement de modo a obtermos um apontador XMLElement* da tag B. De forma a obterem-se os valores dos componentes pretendidos realiza-se um Attribute do elemento XML desejado.

Após serem armazenados os modelos a desenhar num vector<string>, foi percorrida essa estrutura de dados e foram abertos os ficheiros 3d correspondentes. Na primeira linha encontra-se o número de linhas que o ficheiro possui, pelo que foi feito um ciclo for em que a cada iteração foi realizado um getline e criado um objeto Point com os 3 componentes x, y e z. A cada iteração, estes pontos foram sendo armazenados num vector<Point>. De seguida, percorre-se essa estrutura de dados e desenha-se na renderscene pontos 3 a 3 com o auxílio de glVertex3f. Todos os pontos no vector<Point> vertex já se encontram ordenados de acordo com regra da mão direita.

Por outro lado, foi ainda definido algumas captações de eventos a partir do teclado (método 'processSpecialKeys'). Relativamente, ao modo de desenho(preenchimento) por default foi definido o glPolygonMode a *GL_LINE*. Ao pressionar a tecla especial *GLUT_KEY_END* a(s) primitiva(s) ficam totalmente preenchidas (*GL_FILL*). Enquanto que ao pressionar a tecla *GLUT_KEY_PAGE_DOWN* a(s) primitiva(s) são representadas em pontos. Para voltar para modo linha, terá de ser pressionada a tecla *GLUT_KEY_PAGE_UP*.

Nesta fase foi ainda implementado captura de eventos (método 'processKeys') e conseqüente movimentação da câmara. Para uma dada tecla ('a','d','w','s','q' ou 'e') é alterado o valor do ângulo da câmara em relação a um eixo o que resulta na alteração da posição da mesma.

4. Manual de Utilização

De modo a gerar os modelos desejados, é necessário executar o generator com pelo menos 4 parâmetros. O primeiro é o nome do executável "generator", de seguida o nome da primitiva (sphere, box, cone, plane, cylinder ou prisma) e por fim o nome do ficheiro onde serão armazenados todos os pontos do modelo. Depois dependendo da primitiva desejada os parâmetros intermédios serão diferentes para cada um.

box: comprimento do lado, seguido do número de divisões da grelha

sphere: raio, seguido do número de fatias verticais(slices), e das fatias horizontais(stacks)

cone: o raio da base, a altura, as slices e por fim o número de stacks

plane: o comprimento dos lados, seguido do número de divisões da grelha

cylinder: raio da base e do topo, a altura, os slices e o número de stacks.

prisma: raio da base, raio do topo, a altura, os slices e o número de stacks.

De forma a apresentarmos os modelos desejados temos que mencioná-los no ficheiro de configuração XML. Ficheiro este que será passado como argumento ao executar a engine. Uma limitação é que o ficheiro tem de estar ou na current working directory ou dentro de um folder na current working directory. Opcionalmente, poderá não ser passado nenhum ficheiro de configuração como argumento, sendo que por default será considerado o ficheiro "config.xml" apresentado de seguida.

```
<?xml version="1.0" encoding="utf-8"?>
<world>
  <window width="512" height="512" />
  <camera>
    <position x="8" y="4" z="4" />
    <lookAt x="0" y="0" z="0" />
    <up x="0" y="1" z="0" />
    <projection fov="60" near="1" far="1000" />
  </camera>
  <group>
    <models>
      <model file="plane.3d" />
      <model file="sphere.3d" />
    </models>
  </group>
</world>
```

Figura 4.1: Ficheiro de configuração XML por default

5. Testes

Foram executados os vários testes providenciados pelos docentes, sendo que o resultado dos mesmos foi o expectável, como é possível averiguar pelas imagens seguintes. Para além das primitivas que foram enunciadas, complementámos também com a criação de um prisma e um cilindro (fig.5.6 e 5.7 correspondentemente).

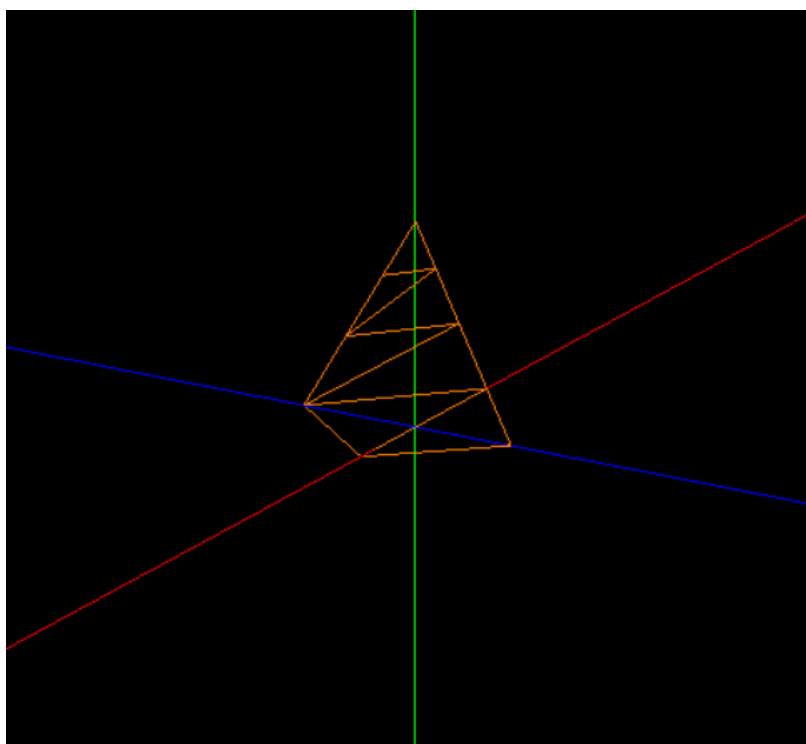


Figura 5.1: Resultado de test_1_1.xml



Figura 5.2: Resultado de test_1_2.xml

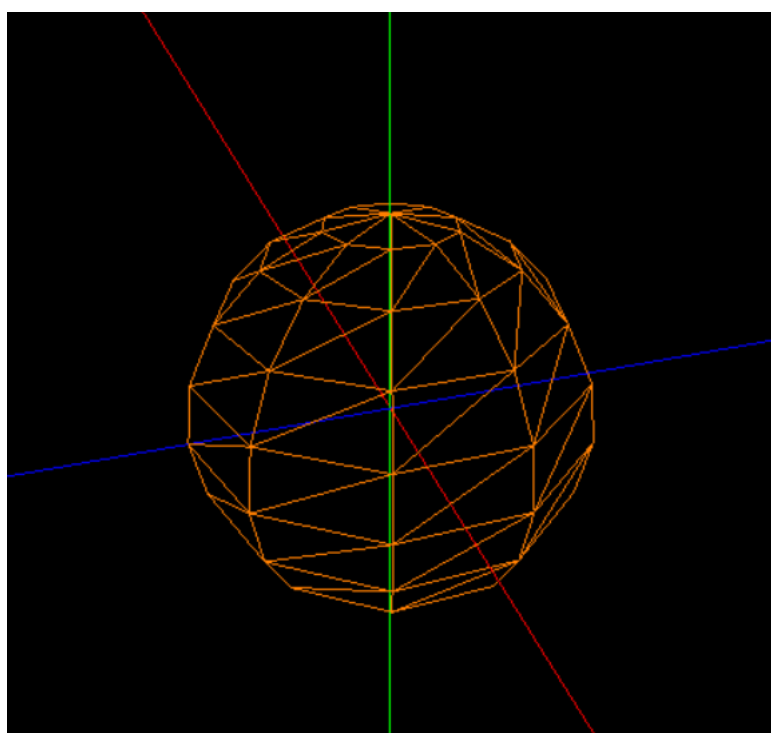


Figura 5.3: Resultado de test_1_3.xml

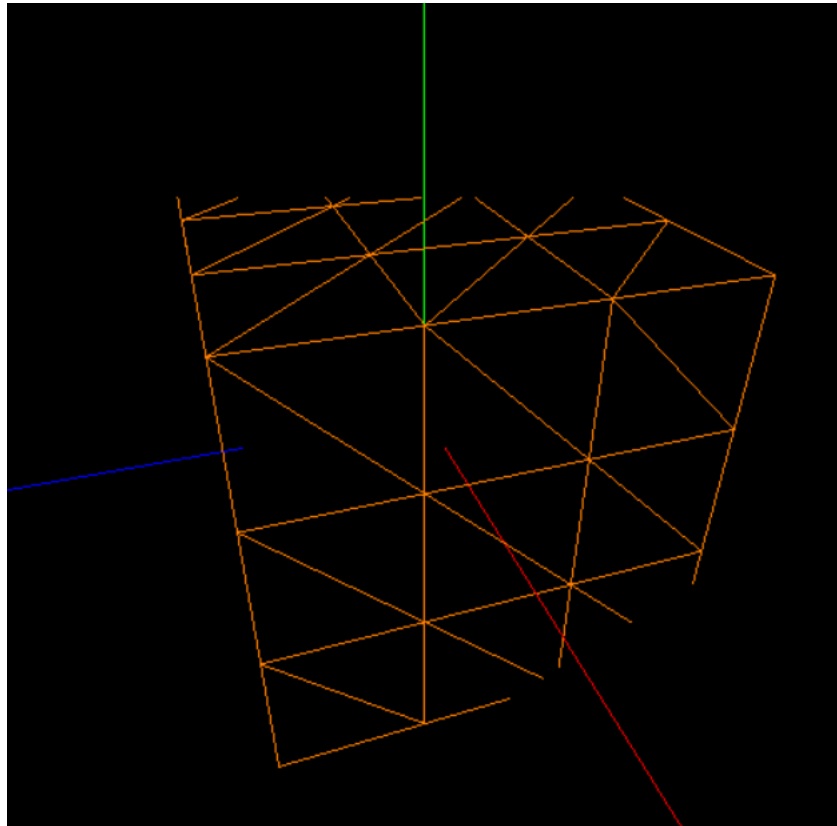


Figura 5.4: Resultado de test_1_4.xml

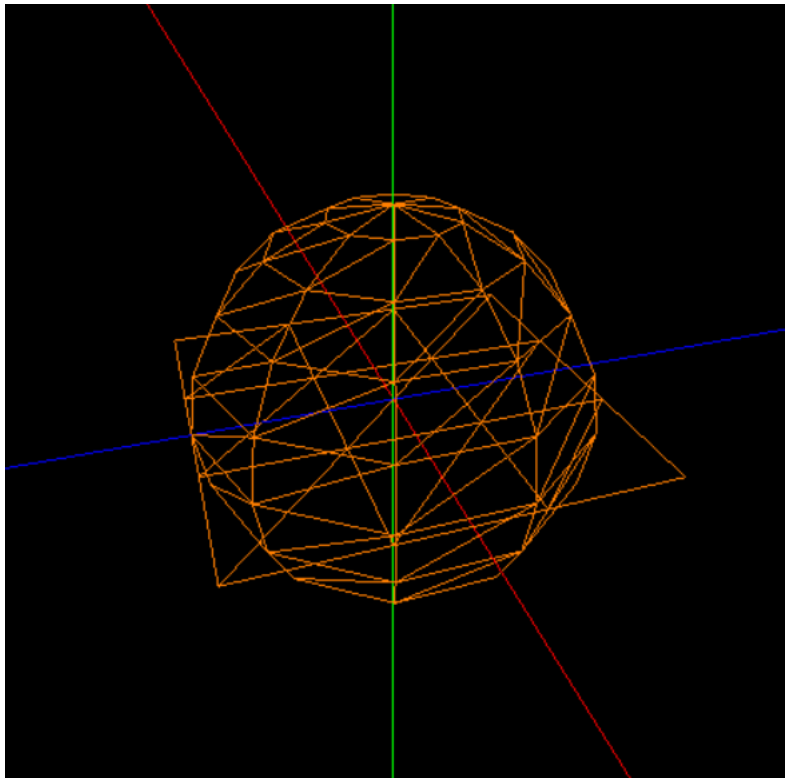


Figura 5.5: Resultado de test_1_5.xml

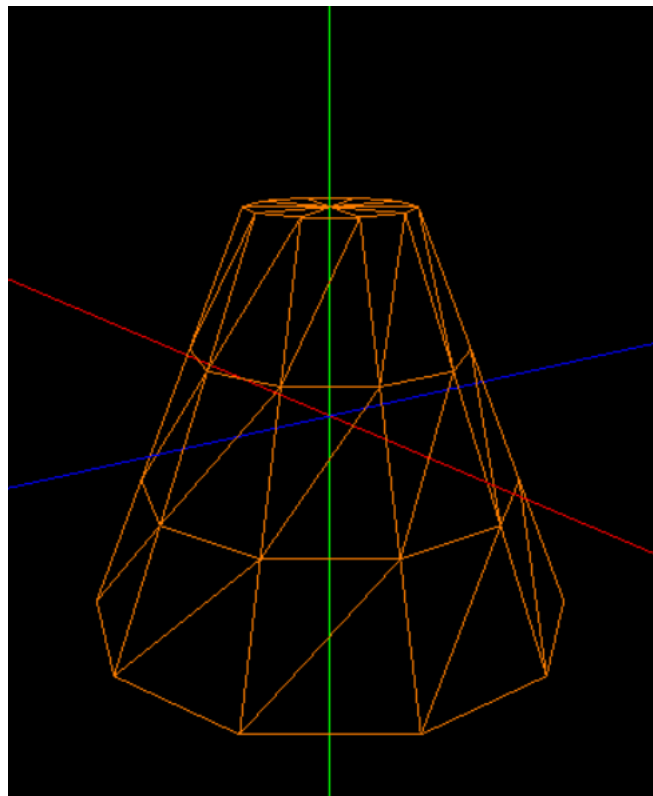


Figura 5.6: Prisma

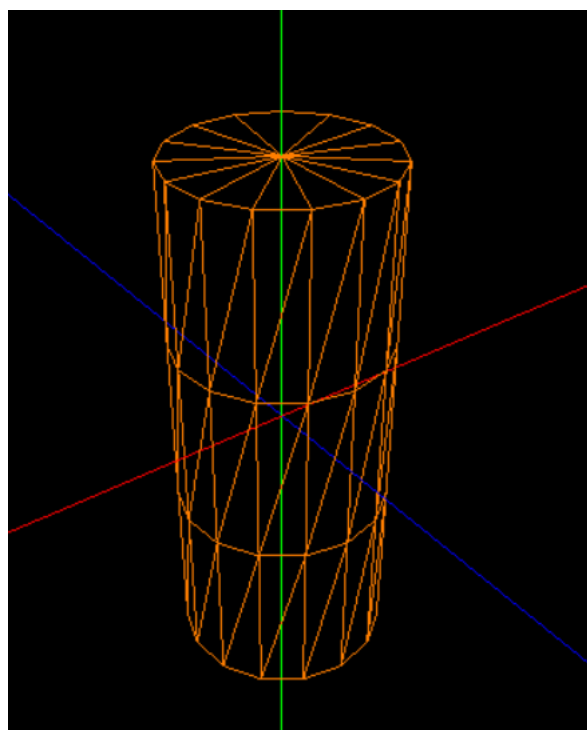


Figura 5.7: Cilindro

6. Conclusão

Durante a primeira fase do projeto, o grupo foi desafiado a usar os conhecimentos adquiridos nas primeiras semanas da UC para criar as formas geométricas desejadas pelos professores. Esta atividade permitiu que o grupo praticasse a manipulação de vértices e triângulos para melhor criar formas geométricas.

Familiarizando-se com esses conceitos, o grupo sentiu-se mais confortável e confiante em relação às próximas fases do projeto. Com um entendimento mais sólido da manipulação de vértices e triângulos, o grupo poderá concentrar-se em aprimorar as técnicas e na criação de formas mais complexas e detalhadas no futuro.

Além disso, o grupo também aprendeu a importância da colaboração e do trabalho em equipa na resolução de problemas e na criação de soluções eficazes.