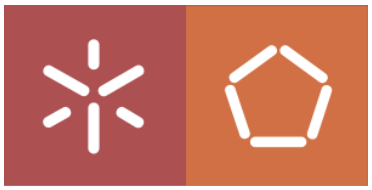

Universidade do Minho
Escola de Engenharia

Universidade do Minho
Mestrado Integrado em Engenharia Informática

Computação Gráfica

Projeto Computação Gráfica
1ª Fase

6 Março 2020

Ana Margarida Campos
(A85166)

Ana Catarina Gil
(A85266)

Tânia Rocha
(A85176)

Contents

List of Figures

Introdução

1.1 Contextualização

O presente relatório foi elaborado no âmbito da UC de Computação Gráfica. É pretendido com o mesmo, formalizar e modelar toda a análise envolvida na construção da solução do enunciado proposto pelos docentes da UC.

Para a primeira fase deste projeto foi nos requerido duas aplicações distintas, sendo a primeira a geração de ficheiros com informação resultante da elaboração de vértices de acordo com os argumentos dados (gerador) e outra que permite visualizar os modelos a partir desses mesmos ficheiros identificados a partir de um ficheiro XML (engine).

Para a implementação destas duas aplicações,utilizando a linguagem de programação C++, é demostrado ao longo deste relatório as resoluções formais e matemáticas para a geração dos diferentes sólidos.

1.2 Descrição do Enunciado

Para o gerador de vértices é necessário que este seja capaz de gerar as seguintes primitivas, de acordo com os seguintes argumentos:

- **Plano** - Através da sua dimensão;
- **Caixa** - Através das dimensões em X, Y e Z e o nº de divisões desejadas;
- **Esfera** - Através do raio, slices e stacks;
- **Cone** - Através do raio da base, altura, slices e stacks.

Estrutura do código

2.1 Generator

No *Generator* é apresentado o código para gerar os vértices, constituídos por três coordenadas x, y e z, de cada uma das figuras pretendidas para esta fase do trabalho, em que cada sequência de três vértices representará um triângulo. Neste ponto da aplicação terá de ser tido em conta a *Regra da mão direita* de maneira a permitir a contrução correta dos triângulos.

Uma vez gerados estes irão ser armazenados em ficheiros *.3d*. Nestes, em cada linha existem 3 floats que representam um vértice. Os floats aparecem por ordem de eixo x, y, z.

2.2 Engine

O *Engine* tem como principal objetivo a leitura dos ficheiros *.3d* e a renderização dos mesmos. A leitura dos ficheiros de configuração serão levadas a cabo dependendo do conteúdo presente no ficheiro XML (*scene.xml*), que é intrepertado através de uma biblioteca externa *TinyXML2*. Este conteúdo especifica qual ou quais os ficheiros *.3d* que deverão ser lidos.

Para ser possível visualizar as figuras foi utilizada a API *OpenGL*.

```
0.500000 0.000000 0.500000
0.500000 0.000000 -0.500000
-0.500000 0.000000 0.500000
-0.500000 0.000000 -0.500000
-0.500000 0.000000 0.500000
0.500000 0.000000 -0.500000
```

Figure 2.1: Exemplo de ficheiro .3d

.

Resolução Formal do Problema e Implementação

Todos os cálculos e equações feitas nestas secção, serão utilizados na parte de implementação para a modelagem das figuras.

3.1 O Plano

Como descrito anteriormente, para a criação de um plano, é utilizado como parâmetro o comprimento do lado, isto porque o plano irá tratar-se de um quadrado centrado na origem. Este terá de ser construído através da geração de dois triângulos.

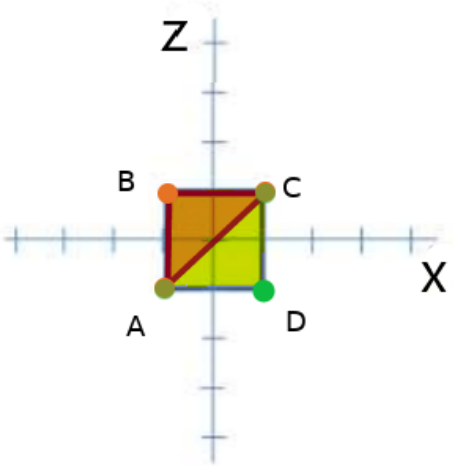

Figure 3.1: Contrução de um plano com triângulos

Como é possivel observar, as coordenadas x e y dos vértices possuem o mesmo valor absoluto e o eixo Y terá as suas coordenadas a 0. Considerando *lado* como o comprimento de um dos lados, obtemos os seguintes vértices :

- Ponto A

$$(-lado/2, 0, -lado/2) \tag{3.1}$$

- Ponto B

$$(-lado/2, 0, lado/2) \tag{3.2}$$

- Ponto C

$$(lado/2, 0, lado/2) \tag{3.3}$$

- Ponto D

$$(lado/2, 0, -lado/2) \tag{3.4}$$

3.1.1 Implementação

De maneira a construir o plano desenvolvemos a função **void plane(float largura, string filename)** através do algoritmo anteriormente apresentada. Após se definir a largura pretendida, os pontos dos triângulos criados vão ser armazenados no ficheiro passado como argumento (por exemplo **plano.3d**) para de seguida serem lidos e renderizados. O resultado da renderização é visto na figura seguinte:

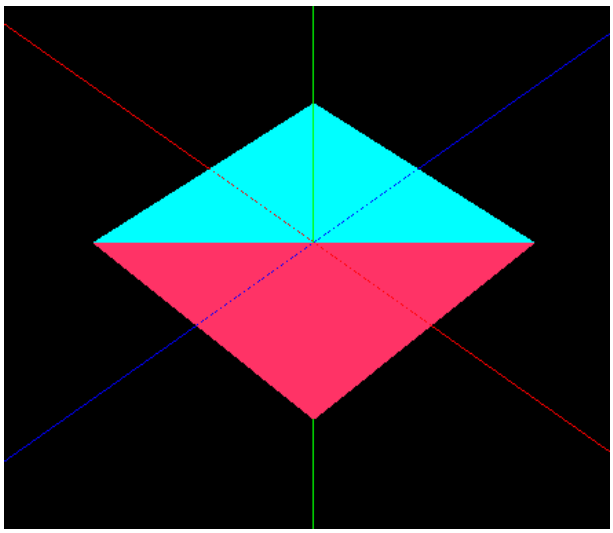

Figure 3.2: Plano

3.2 A Caixa

Para a criação da caixa, foram utilizados vários argumentos: o número de divisões, o comprimento, a altura e a largura da caixa.

As divisões da caixa são o número de partes que uma face está dividida. Tendo isto em conta, se o número de divisões escolhido for, por exemplo, 3,então nessa face teremos 9 quadriláteros. Ou seja, o número de quadrilateros por face é

$$divisões^2 \tag{3.5}$$

Comecemos por definir o número de divisões desejadas por:

$$NDivisões \tag{3.6}$$

Se considerarmos então que a caixa tem x de lado, y de altura e z de largura, as dimensões de cada quadrilátero estarão relacionadas com o tamanhho da face em questão e o número de divisões da mesma. Representaremos esse tamanho por p:

$$px = x/NDivisões \tag{3.7}$$

$$py = y/NDivisões \tag{3.8}$$

$$pz = z/NDivisões \tag{3.9}$$

Nota adicional: O centro da caixa é considerado a origem do referencial onde o modelo é construído, dai as divisões serem representavéis desta maneira.

Passando à construção dos triângulos necessários para construir a caixa nas condições descritas foi utilizado o modelo a seguir apresentado:

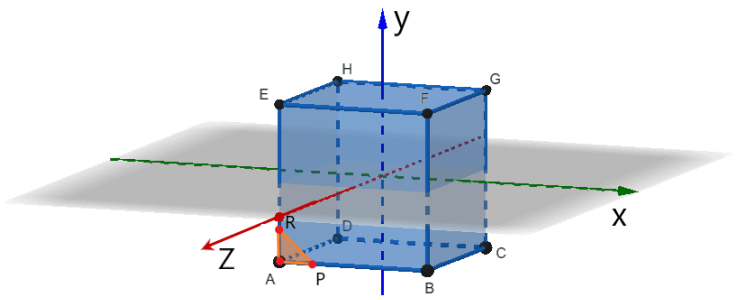

Figure 3.3: Caixa centrada na origem

Para poder definir um triângulo numa das faces, deve ser utillizado uma sequência de vértices representada na figura anterior, cujos pontos são calculados da seguinte forma (face frontal):

- **Ponto A**

$$(px * i - x/2, py * q - y/2, z/2) \tag{3.10}$$

- **Ponto P**

$$(px * i - x/2 + px, py * q - y/2, z/2) \tag{3.11}$$

- **Ponto R**

$$(px * i - x/2, py * q - y/2 + py, z/2) \tag{3.12}$$

Nota: Consideramos que i e q variam entre 0 e o número de divisões para assim ser possível desenhar todos os triângulos necessários.

3.2.1 Implementação

De maneira a construir a caixa desenvolvemos a função **void box(float x, float y, float z, int nDivisoes, string filename))** através do algoritmo anteriormente apresentado. Após se definir o comprimento do lado, a altura, a largura e o número de divisões pretendidos, os pontos dos triângulos criados vão ser armazenados no ficheiro passado como argumento (por exemplo **box.3d**) para de seguida serem lidos e renderizados. O resultado da renderização é visto na figura seguinte:

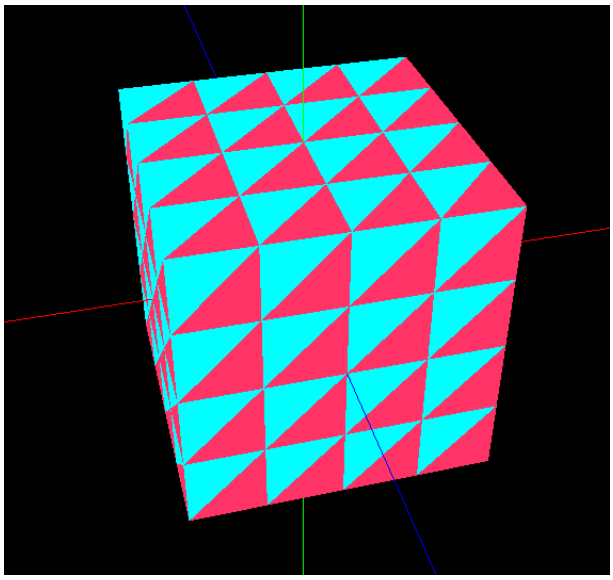

Figure 3.4: Caixa

3.3 A Esfera

Como descrito na introdução, para obter as 3 coordenadas de um vértice da esfera, são necessários 3 parâmetros:

- Raio da Esfera
- Slice
- Stack

Começando pela nomenclatura: uma *slice* e uma *stack* representam divisões orientade de acordo com a seguinte figura:

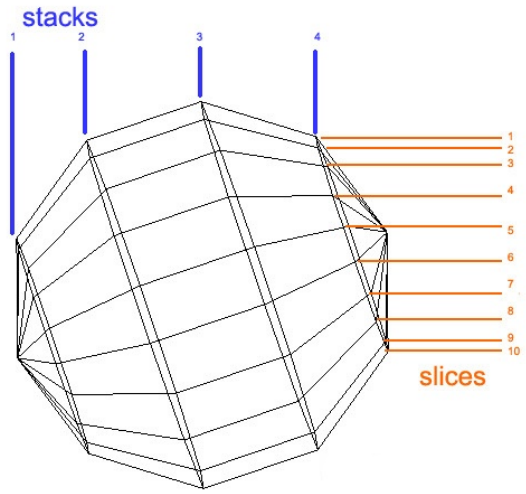

Figure 3.5: Representação de slices e stacks

Para auxiliar a elaboração da esfera críamos dois ângulos distintos:

- α que é definido por

$$\alpha = 2\pi/slices \tag{3.13}$$

- β que é definido por

$$\beta = \pi/stacks \tag{3.14}$$

O ângulo α representa a rotação em radianos a partir do eixo positivo do x em relação ao eixo do y, no sentido contrário dos ponteiros do relógio e toma valores entre 0 e 2π.

O ângulo β representa a rotação em radianos a partir do eixo positivo do y em relação ao eixo do z, no sentido dos ponteiros do relógio e toma valores entre 0 e π.

De seguida está representada uma esfera centrada na origem do referencial com a representação de ambos os ângulos.

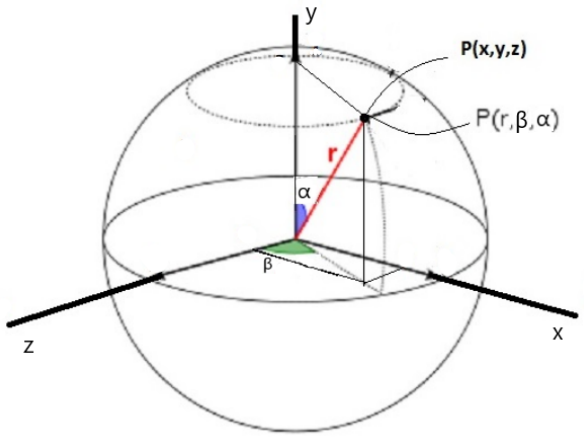

Figure 3.6: Esfera centrada na origem

De maneira a representar os vértices numa esfera tivemos de recorrer à conversão de coordenadas cartesianas para coordenadas polares através das seguintes equações:

- Calcular x :

$$x = r * cos(\beta) * sin(\alpha) \tag{3.15}$$

- Calcular y :

$$y = r * sin(\beta) \tag{3.16}$$

- Calcular z :

$$z = r * cos(\beta) * cos(\alpha) \tag{3.17}$$

Podemos agora, após termos definido como se calculam os pontos à superficie da circunferência, calcular os pontos necessários para obter triângulos de forma a ser possível representar uma esfera.

Na figura seguinte, são representados 4 pontos necessários para a criação de dois triângulos (superior e inferior).

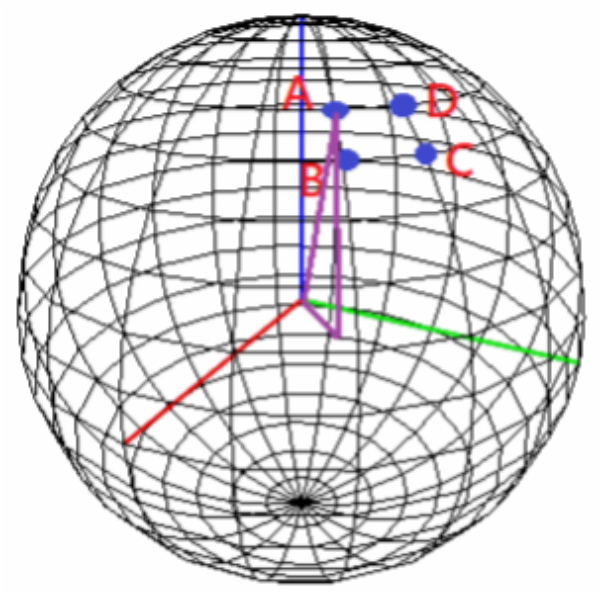

Figure 3.7: Esfera centrada na origem

O triângulo superior é composto pelos pontos ... e o triângulo inferior pelos pontos ... De acordo então com a figura,estes mesmos pontos podem ser calculados através das seguintes equações:

- **Ponto A**

$$x = r * cos(q * \beta) * sin(i * \alpha) \tag{3.18}$$

$$y = r * sin(q * \beta) \tag{3.19}$$

$$z = r * cos(q * \beta) * cos(i * \alpha) \tag{3.20}$$

- **Ponto B**

$$x = r * cos(q * \beta) * sin(i * \alpha) \tag{3.21}$$

$$y = r * sin((q + 1) * \beta) \tag{3.22}$$

$$z = r * cos((q + 1) * \beta) * cos(i * \alpha) \tag{3.23}$$

- **Ponto C**

$$x = r * cos((q + 1) * \beta) * sin((i + 1) * \alpha) \tag{3.24}$$

$$y = r * sin((q + 1) * \beta) \tag{3.25}$$

$$z = r * cos((q + 1) * \beta) * cos(i * \alpha) \tag{3.26}$$

- **Ponto D**

$$x = r * cos(q * \beta) * sin((i + 1) * \alpha) \tag{3.27}$$

$$y = r * sin(q * \beta) \tag{3.28}$$

$$z = r * cos(q * \beta) * cos((i + 1) * \alpha) \tag{3.29}$$

Nota: i e q variam entre 0 e o número de *slices* e *stacks*, respetivamente, para permitir a construção de toda a esfera.

3.3.1 Implementação

De maneira a construir a esfera desenvolvemos a função **void sphere(float r, int slices, int stacks, string filename)** através do algoritmo anteriormente apresentado. Após se definir o raio e o número de slices e stacks pretendidos, os pontos dos triângulos criados vão ser armazenados no ficheiro passado como argumento (por exemplo **sphere.3d**) para de seguida serem lidos e renderizados. O resultado da renderização é visto na figura seguinte:

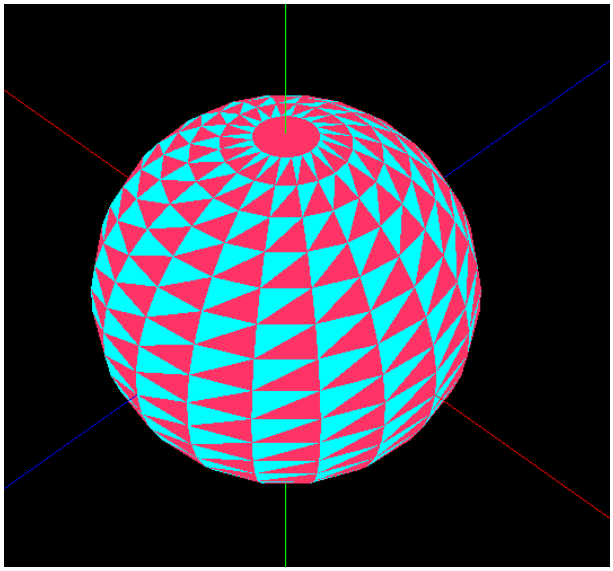

Figure 3.8: Esfera

3.4 O Cone

No caso da criação do cone, foram utilizados 4 parâmetros:

- O raio da base
- A altura
- O número de slices
- O número de stacks

No entando neste caso, terá de ser construído duas formas distintas: a base, que é um círculo, e a superfície lateral do cone.

3.4.1 Base do Cone

Sendo a base de um cone uma circunferencia, temos que ter em conta o número de *slices* fornecidos. Aproveitando o angulo α da esfera:

$$\alpha = 2\pi/slices \tag{3.30}$$

Com o ângulo entre os vértices e o raio da circunferência, conseguimos calcular as coordenadas dos vértices, recorrendo às coordenadas polares.

O cálculo da coordenada em x e em z de um vértice é dado pelas expressões:

$$x = raio * cos(\alpha) \tag{3.31}$$

$$z = raio * sin(\alpha) \tag{3.32}$$

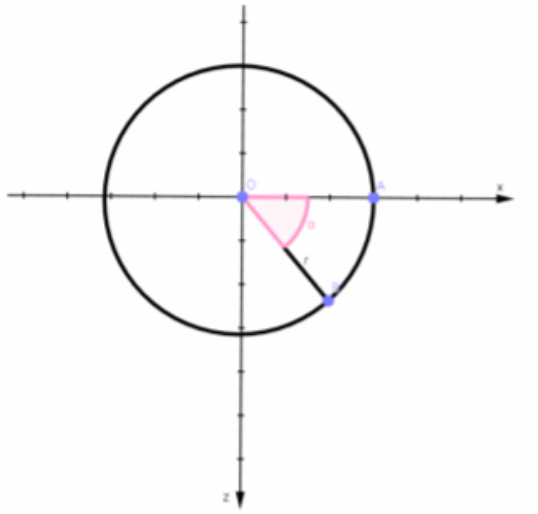

Figure 3.9: Circunferência centrada na origem

Tendo em conta que a base vai ser contruída no plano **xz**, o eixo de Y tem sempre o valor de 0 para todos os vértices da base.

Seguindo os pontos marcados na figura, serão definidas as expressões para calcular os mesmos:

Temos para já o Ponto O, a origem, com coordenadas

- **Ponto O**

$$(0, 0, 0) \tag{3.33}$$

- **Ponto A**

$$(raio * cos(i * \alpha), 0, raio * sin(i * \alpha)) \tag{3.34}$$

- **Ponto B**

$$(raio * cos((i + 1)\alpha), 0, raio * sin((i + 1)\alpha)) \tag{3.35}$$

Nota: i varia entre 0 e o número de slices.

3.4.2 Superfície Lateral do Cone

Quanto à superficie lateral do cone, a estratégia adotada foi gerar uma sequência de par de triângulos por stack na mesma quantidade que os *slices* da base. Para cada triângulo, as coordenadas dos vértices que se encontram no plano XoZ são os mesmos dos que foram definidos na base, respeitando assim o número de *slices*. Podemos observar a figura seguinte como exemplo, que será utilizada também para o cálculo exemplar dos vértices dos quadriláteros mais à frente:

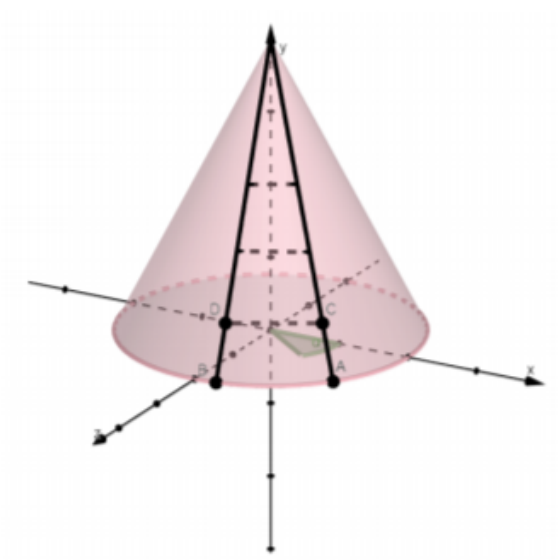

Figure 3.10: Cone com base centrada na origem

Podemos calcular o tamanho de cada *stack*, através da altura do cone e do número de *stacks*:

$$alturaStack = (altura/stacks) * q \tag{3.36}$$

Para possibilitar o cálculo dos vértices dos quadriláteros, para além da variação do x temos também a variação de y e z. Além destas alterações constantes temos também a alteração do raio. A expressão resultante destes requisitos é:

$$raioSeguinte = (altura - alturaStack) * raio/altura \tag{3.37}$$

No cone o raio vai diminuindo à medida que a altura aumenta. No caso da geração do triângulo ABD, como o raio em A e B é maior que o raio em D, foi necessária a criação de uma nova variável ao qual nomeamos *raioSeguinte2*. Do mesmo modo, como a alturaStack difere foi também necessária a criação de outra variàvel denominada *alturaSatck2*.

$$raioSeguinte2 = (altura - alturaStack2) * (raio/altura) \tag{3.38}$$

$$alturaStack2 = (altura/stacks) * (q + 1) \tag{3.39}$$

As coordenadas genéricas para a criação dos triângulos são:

$$x = raio * cos(i * \alpha) \tag{3.40}$$

$$y = altura \tag{3.41}$$

$$z = raio * sin(i * \alpha) \tag{3.42}$$

Nota: i e q varia entre 0 e o número de *slices* ou *stacks* respetivamente.

3.4.3 Implementação

De maneira a construir o cone desenvolvemos a função **void cone (float raio,float altura, int slices, int stacks,string filename)** através do algoritmo anteriormente apresentado. Após se definir o raio, a altura e o número de slices e stacks pretendidos, os pontos dos triângulos criados vão ser armazenados no ficheiro passado como argumento (por exemplo **cone.3d**) para de seguida serem lidos e renderizados. O resultado da renderização é visto na figura seguinte:

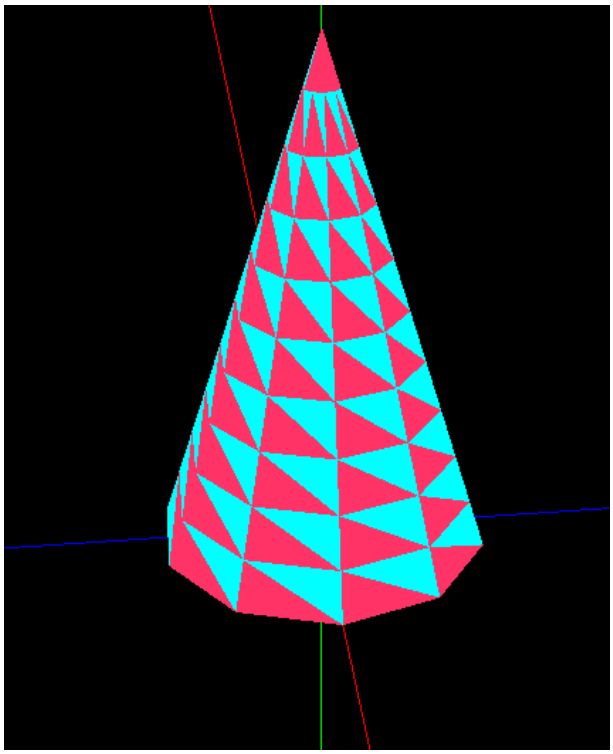

Figure 3.11: Cone

Conclusão

Com o desenvolvimento da primeira parte do projeto foi possível consolidar conhecimentos obtidos nas aulas práticas e teóricas desta unidade curricular.

Esta fase consistiu essencialmente em construir figuras geométricas através de determinadas ferramentas que ainda não tinhamos tido a oportunidade de utilizar, nomeadamente, ficheiros do tipo *XML*, a API *OpenGL*, a linguagem de programação C++, etc.