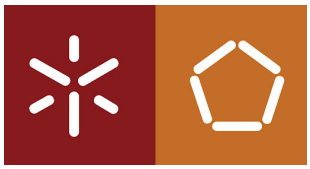
Universidade do Minho

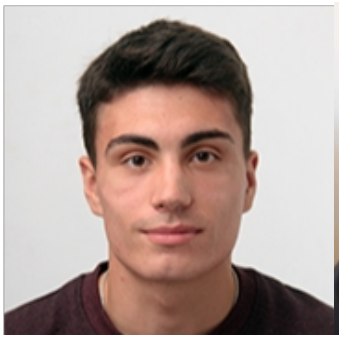
Departamento de Informática

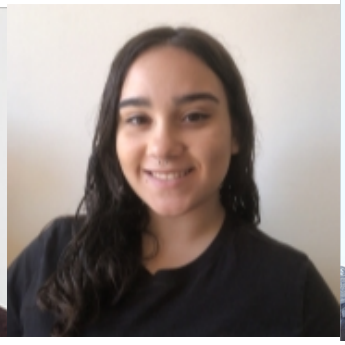


licenciatura em engenharia informática

Computação gráfica - 1ª fase

Grupo 38


Henrique Pereira


Mariana Morais


Ana João


Simão

Henrique Morais Pereira A100831
Mariana Filipa Morais Gonçalves A100662
Ana João A95128
Simão A100597

Março de 2024

Conteúdo

1 Introdução

A unidade curricular de Computação Gráfica propõe, recorrendo à biblioteca GLUT, a utilização do OpenGL para a construção de modelos 3D com recurso à linguagem de programação C++. 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.

Com a criação dos modelos propostos no enunciado pretendemos consolidar os conceitos abordados nas aulas iniciais.

Esta fase será dividida em dois pontos fulcrais, um *Generator* que gera ficheiros com os pontos necessários para a criação dos modelos desejados e um *Engine* que lê um ficheiro de configuração, escrito em XML, e representa graficamente os modelos.

Posto isto, temos de forma muito breve o assunto do presente relatório que se refere à fase 1 de um projeto dividido em 4 fases, tal como indicado pela equipa docente.

2 Descrição do Problema

2.1 Fase 1

A primeira fase deste projeto tem como intuito a criação de dois programas: um Gerador que gera um ficheiro com os pontos do modelo especificado, e um Motor que lê um ficheiro de configuração, escrito em XML, e representa graficamente os modelos. Como foi dito anteriormente, todas as fases do projeto possuem requisitos que deverão ser cumpridos. Posto isto, os requisitos estipulados para esta primeira fase são:

2.1.1 Generator

O *Generator* irá receber como parâmetros o tipo da primitiva gráfica, parâmetros relativos a cada modelo e o nome do ficheiro onde serão guardados o número de pontos para desenhar a primitiva e as coordenadas de todos os pontos necessários para a sua concessão. Tendo como requisito a criação de 4 primitivas gráficas:

- plane side file.3d: quadrado no plano XZ, centrado na origem, subdividido nas direções X e Z.
- box length divisions file.3d: necessita das dimensões x, y e z e do número de divisões para cada lado, também centrada na origem.
- sphere radius slices stacks file.3d: necessita de raio e número de divisões verticais e horizontais, igualmente centrada na origem.
- cone radius height slices stacks file.3d: necessita do raio da base, da altura e do número de divisões verticais e horizontais, sendo a parte inferior do cone no plano XZ.

E além disto, como já referido, este tem de armazenar os vértices dos modelos num ficheiro.3d.

2.1.2 Engine

O *Engine* tem como requisito os seguintes pontos:

- ler o ficheiro XML onde estão os modelos das primitivas.
- armazenar os vértices dos modelos em memória.
- desenhar, utilizando triângulos, as primitivas relativas aos modelos desejados.

3 Resolução do Problema

Visto que o trabalho irá focar-se essencialmente na representação de vértices num plano com 3 dimensões, de modo a agilizar o processo, construimos a seguinte estrutura:

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

Figura 1: Struct Point

Esta estrutura pretende representar as coordenadas x,y,z de um ponto.

3.1 Generator

3.1.1 Plano

O plano será formado por vários pontos, sendo calculado pelo número de divisões efetuadas no mesmo. Sendo que cada triângulo é formado por 3 vértices e cada quadrado é a junção de 2 triângulos com vértices coincidentes. Cada quadrado será formado por 6 pontos. O nosso plano será formado por várias divisões, divisões estas que irão formar quadrados. Ou seja, se o nosso número de divisões for 3, teremos 3*3 quadrados. E como cada quadrado são 2 triângulos, teremos 3*3*2 triângulos. Sendo a fórmula do número de quadrados= divisões*divisões*2.
Logo teremos 3(divisões)*3(divisões)*6(pontos de cada quadrado) = 54 pontos

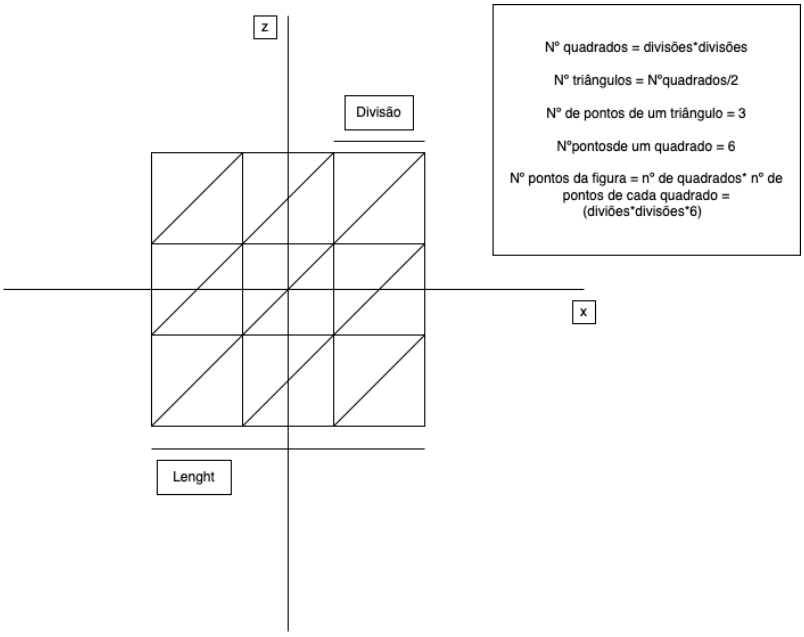


Figura 2: Pontos do plano

O plano centrado na origem e sobre os eixos x e z possuem pontos com 0 na coordenada y.
Antes de falar sobre as restantes coordenadas, é necessário ter a noção de espaçamento (*spacing*) que se refere ao $\frac{lenght}{divisions}$, ou seja, o espaçamento entre cada ponto.

O primeiro ponto (canto inferior esquerdo) tem como coordenada X metade do comprimento total em módulo $\frac{-lenght}{2}$ e posteriormente será adicionando a posição relativa $i * spacing$. A coordenada Z é calculada de maneira semelhante, mas usando a posição relativa $j * spacing$.

A variável i está responsável pela criação das "colunas"e a j pelas "linhas".

O ponto 3 é o canto inferior direito do quadrado. A coordenada X é igual à do ponto 1 e a coordenada Z é calculada da mesma forma que point1, mas usando $float(j+1) * spacing$. Isso cria um ponto na mesma linha do ponto 1.

O ponto 2 é o canto superior esquerdo do quadrado. A coordenada X é calculada da mesma forma que point1, mas usando $float(i+1) * spacing$. Sendo a coordenada Z igual à do ponto 1. Isto cria um ponto acima do point1, mas na mesma coluna.

O ponto 4 é o canto superior direito do quadrado. A coordenada X é calculada usando $float(i+1) * spacing$ e a coordenada Z é calculada usando $float(j+1) * spacing$.

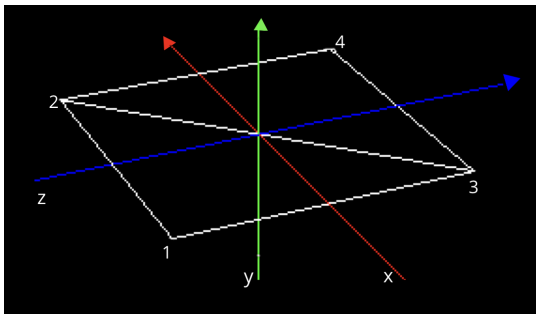
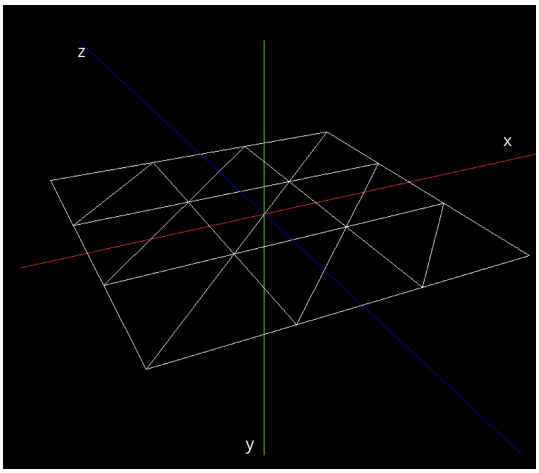
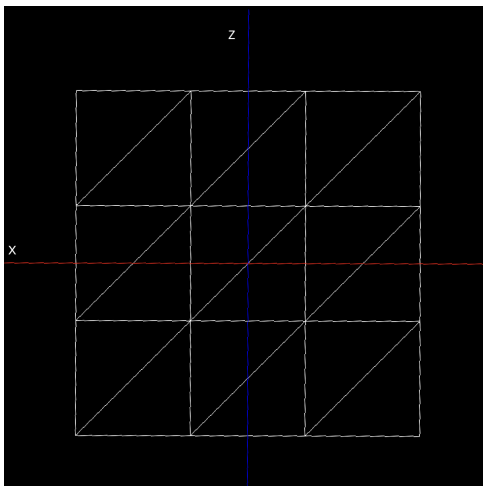


Figura 3: Pontos

Após isto, podemos verificar que com a regra da mão direita obtemos um plano orientado no sentido positivo do eixo y. Isto consegue-se desenhando os triângulos formados pelos pontos representados na figura, pela seguinte ordem 1, 3, 2 e 4, 2, 3.



Vista de frente



Vista de cima

3.1.2 Caixa

De modo a fazer a contagem de pontos, semelhante ao plano, e vendo o cubo como a junção de 6 planos, pois este tem 6 faces, o raciocínio assemelha-se. O número de pontos da figura será o número de pontos de 1 das faces

multiplicando por 6. O número de pontos da face, tal como vimos para um plano será dado por divisões*divisões que nos dará o número de quadrados possuídos pela nossa face. Este valor multiplicado por 2 dá o número de triângulos, ou seja divisões*divisões*2 que, como cada triângulo tem 3 pontos, ficará cada face com divisões*divisões*2*3 pontos.

Como referimos acima, isto apenas se refere ao número de pontos de uma face, por isto este valor será multiplicado por 6 (faces). Tendo resumidamente, a expressão o seguinte aspeto: *((divisions * divisions) * 6) * 6* .

A nossa implementação resulta na divisão da caixa por 6 faces denominadas por *top, bottom, left side, right side, front* e *back*. As imagens seguintes visam suportar a demonstração da estratégia utilizada.

Tendo em consideração que a base do cubo estará centrada na origem, tal como um plano, o cálculo das coordenadas será igual à dos pontos de um plano. Com a exceção do eixo Y, que será variável entre $\frac{length}{2}$ e - $\frac{length}{2}$, sendo todos os pontos intermédios definidos pelo incremento de - $\frac{length}{2}$ com $\frac{length}{divisions}$ até $\frac{length}{2}$.

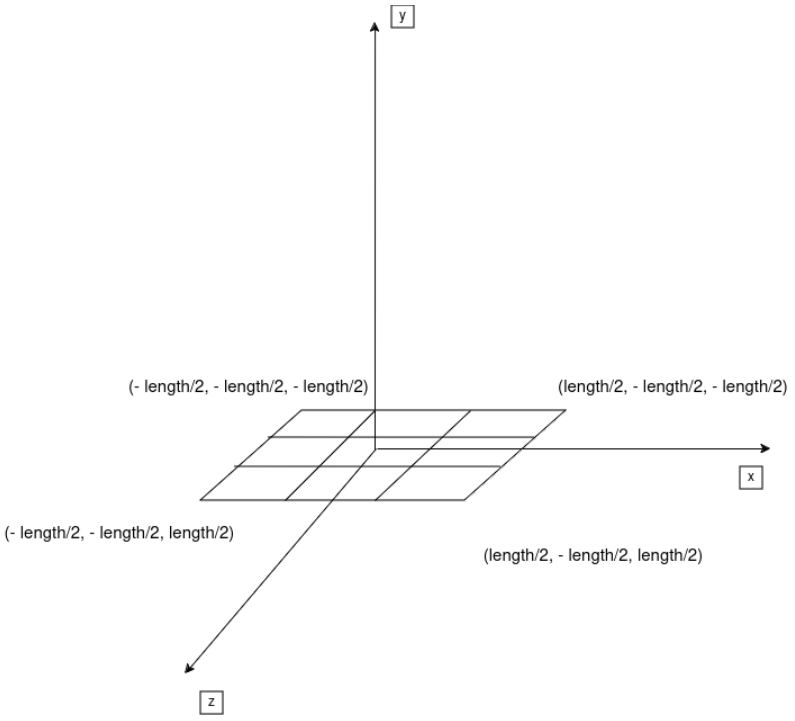


Figura 4: Construção da face bottom

Para o caso da face top, as coordenadas dos pontos são iguais, exceto a coordenada Y, que terá um valor simétrico $\frac{length}{2}$.
Analisando agora a face *back*, os extremos das arestas inferior e superior têm coordenadas tal como mencionadas na figura 5, sendo todos os pontos intermédios definidos pelo incremento de $\frac{-length}{2}$ com $\frac{length}{divisions}$ até $\frac{length}{2}$. Para a face *front*, como é paralela a esta face, as coordenadas do eixo dos X serão todas simétricas, ou seja, iguais a $\frac{length}{2}$.

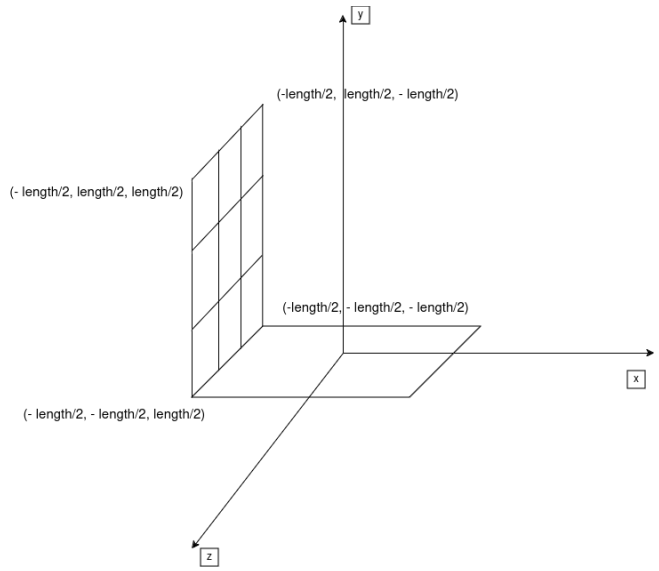


Figura 5: Construção da face back

Por fim, os pontos da face *left side* são calculados da forma anteriormente mencionada. Além disso, a face *right side* será paralela, sendo simétrica a esta relativamente ao eixo do Z.

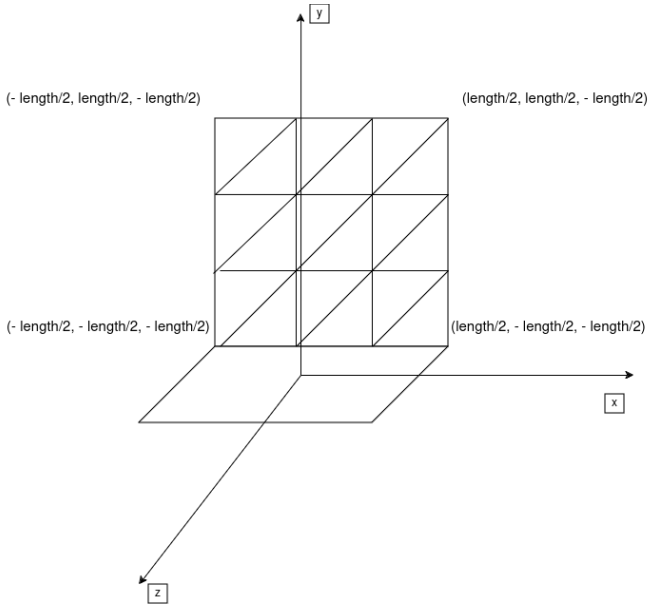


Figura 6: Construção da face left side

Algo a referir que aqui teria um impacto positivo é o uso de inteiros para os ciclos, evitando arredondamentos que iriam resultar no desencontro das faces nos pontos de união. Relativamente a este ponto, é também importante referir que, seria bom termos otimizado o cálculo destes pontos, pois existem pontos que estão a ser recalculados.

3.1.3 Esfera

Para criar uma esfera, usamos um sistema de coordenadas esféricas. Este sistema usa três componentes para definir a posição de um ponto no espaço: O raio, ou seja, a distância do ponto à origem (centro da esfera), o ângulo alpha, ou seja o ângulo horizontal medido a partir de um eixo de referência no plano xy e o ângulo beta que é o ângulo vertical medido a partir do eixo z até o ponto.

A esfera é construída usando dois ângulos, alfa e beta, divididos em várias partes iguais, slices e stacks respetivamente.
O ângulo alpha varia de 0 a 2π radianos, representando a rotação ao redor do eixo y. Este ângulo é dividido em partes iguais. Portanto, cada slice representa um ângulo de slices 2π radianos. O ângulo beta varia de $\frac{-\pi}{2}$ a $\frac{\pi}{2}$ radianos, representando a rotação ao redor do eixo x. Este ângulo é dividido em partes iguais. Portanto, cada stack representa um ângulo de stacks π radianos. Assim, a esfera é construída percorrendo cada slice e cada stack, calculando os pontos na superfície da esfera usando os ângulos alfa e beta, e conectando esses pontos para formar triângulos. Isso resulta num conjunto de triângulos que aproxima a superfície da esfera.

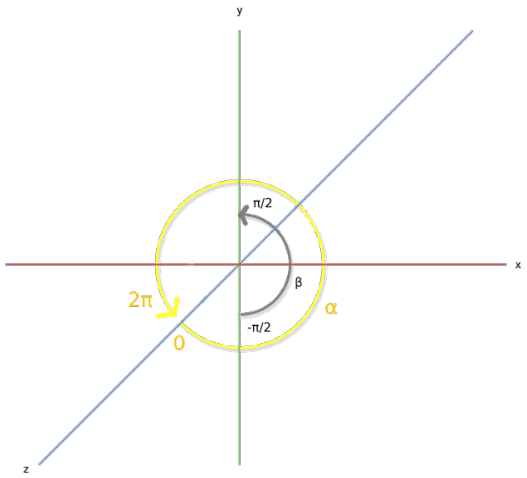


Figura 7: Variação e orientação dos ângulos α e β.

Visto que os ângulos alfa e beta variam para cada ponto, é nos permitido calcular todas as coordenadas dos pontos necessários utilizando as seguintes coordenadas esféricas.

- x = radius×cos(β)×sin(α)
- y = radius×sin(β)
- z = radius×cos(β)×cos(α)

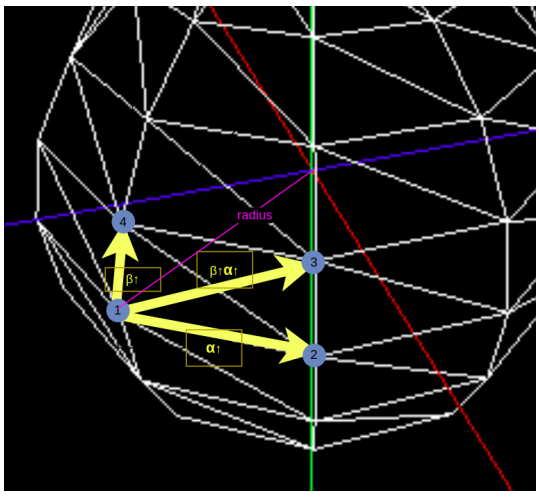


Figura 8: Ordem de cálculo das coordenadas *alpha* e *beta*.

Iniciamos com o triângulo formado pelos pontos 1, 2 e 3, e em seguida o triângulo 3, 4 e 1. Para calcular as coordenadas dos pontos 2, 3 e 4 em relação ao ponto 1, precisamos determinar os valores dos ângulos correspondentes.

Para o ponto 2, relativamente ao ponto 1, apenas o ângulo alpha é alterado (aumenta 1 unidade em relação à divisão atual), para o ponto 3, ambos os ângulos são alterados, e para o ponto 4, é alterado o ângulo beta (aumenta 1 unidade em relação à divisão atual do ângulo). Assim, os ângulos são calculados utilizando as fórmulas:

- $alpha = \frac{2*\pi}{slices} * a$, onde $a \in [0, slices[$
- $beta = \frac{\pi}{stacks} * b$, onde $b \in [\frac{-stacks}{2}, \frac{stacks}{2}[$

Por exemplo, considerando os pontos 1 e 3 da figura 8, se para o ponto 1 $alpha = \frac{2*\pi}{slices} * a$ e $beta = \frac{\pi}{stacks} * b$, então para o ponto 3,$alpha = \frac{2*\pi}{slices} * (a+1)$ e $beta = \frac{\pi}{stacks} * (b+1)$.
Aplicando as coordenadas esféricas com estes valores dos ângulos temos as coordenadas x, y e z de todos os pontos.

3.1.4 Cone

Começando por explicar a base do cone utilizaremos ilustrações dos diferentes cálculos que permitiram calcular as coordenadas dos pontos. Começando pela base, partimos da seguinte ilustração que consideramos bastante útil para uma melhor perceção de como iriam variar ao longo de uma circunferência(base do cone).

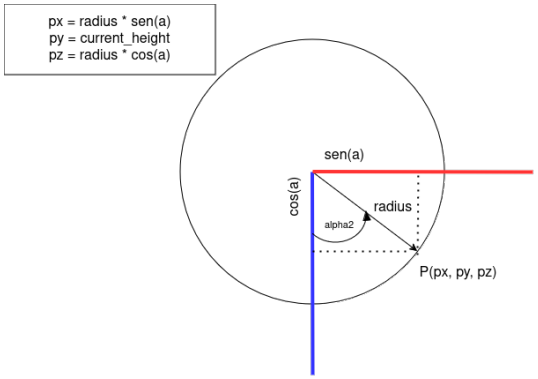


Figura 9: Excerto do formato do ficheiro .3d

Recorrendo a radius (que é alterado aquando da altura), a fórmula da semelhança de triângulos, e ao ângulo α (ângulo formado entre o eixo do cos(a) e radius), podemos calcular as coordenadas x e z, sendo que a coordenada y tem o valor da altura que consideramos no momento.

Estas fórmulas são também utilizadas para calcular os pontos da base do cone fazendo apenas variar o ângulo α de 0 a 2π, tendo a coordenada y o valor 0. Esta variação é feita utilizando a estratégia das divisões. Iniciando na divisão 0 até à divisão *slices* e aumentando $\frac{2\pi}{slices}$ por cada divisão, percorremos todos os valores de α.
Para a base do cone, consideramos então dois ângulos: alpha e alpha2. Assim, o alpha representa o ângulo da posição atual ao longo do círculo da base do cone e alpha2, definido como $alpha + \frac{2\pi}{slices}$, representa o ângulo da próxima posição ao longo do círculo base do cone. Ele é usado principalmente no cálculo das coordenadas x e z dos vértices adjacentes a alpha.

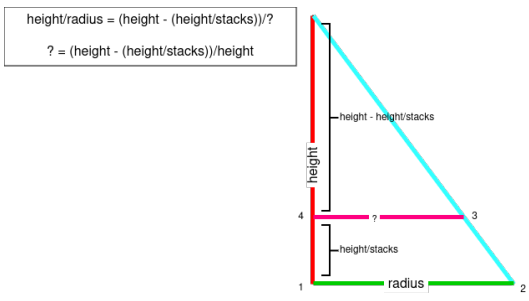


Figura 10: Excerto do formato do ficheiro .3d

O texto descreve a estratégia adotada para desenhar os triângulos que compõem a face do cone. A abordagem consiste em desenhar dois triângulos em cada iteração, formando um quadrado. Assim, seguindo o exemplo dado, começamos desenhando o triângulo formado pelos pontos 1, 2 e 3 e, em seguida, o triângulo 3, 4 e 1.

Para calcular as coordenadas dos pontos 2, 3 e 4 em relação ao ponto 1, precisamos calcular os valores do ângulo e do raio de cada ponto.

- Para o ponto 2, em relação ao ponto 1, apenas o ângulo alpha2 e alpha são alterados, aumentando numa unidade a divisão atual.
- Para o ponto 3, os valores do ângulo e do raio são alterados.
- Para o ponto 4, apenas o raio é alterado, calculado utilizando a fórmula de semelhança de triângulos.
- cone *radius height slices stacks* file.3d: necessita do raio da base, da altura e do número de

Desta forma para os pontos 1 e 3, se para o ponto 1 tivermos $alpha = \frac{2\pi}{slices}$ * j, sendo j um inteiro entre 0 e slices e $alpha2 = \frac{2\pi}{slices} * (j+1)$ e $currentradius = radius * (height - i * (height/stacks))/height$, sendo i um inteiro entre 0 e stacks.

Voltando a considerar, por exemplo, os pontos 1 e 3 da figura se, para o ponto 1 $alpha = \frac{2\pi}{slices} * j$ e $current\_radius = radius * (height - i * (height/stacks))/height$, então para o ponto 3, $alpha2 = \frac{2\pi}{slices} * (j+1)$ e $current\_radius = radius * (height - (i-1) * (height/stacks))/height$.

O processo repete-se até atingir a altura definida previamente.

3.1.5 Toro

Para criar um toro, usamos um sistema de coordenadas cilíndricas. Este sistema usa três componentes para definir a posição de um ponto no espaço: O raio interno (innerRadius), o raio externo (outerRadius), e dois ângulos, alpha e beta.

O toro é construído usando os dois ângulos, alpha e beta, divididos em várias partes iguais, slices e stacks respetivamente.

O ângulo alpha varia de 0 a 2pi radianos, representando a rotação ao redor do eixo y. Este ângulo é dividido em slices partes iguais. Portanto, cada slice representa um ângulo de slices 2π radianos.

O ângulo beta também varia de 0 a 2pi radianos, representando a rotação ao redor do eixo x. Este ângulo é dividido em stacks partes iguais. Portanto, cada stack representa um ângulo de stacks 2π radianos.

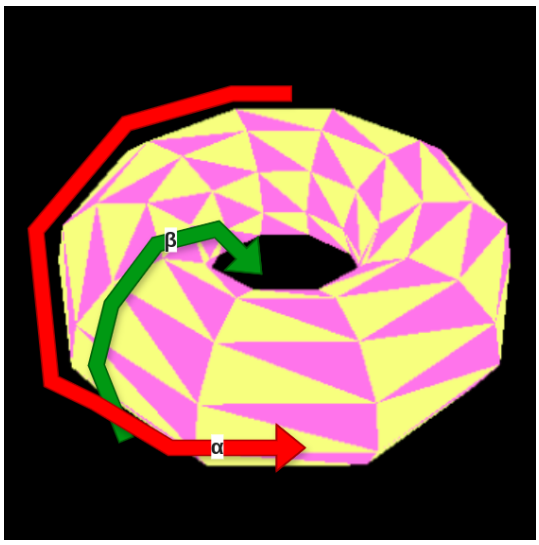


Figura 11: Variação e orientação dos ângulos α e β.

Assim, o toro é construído percorrendo cada slice e cada stack, calculando os pontos na superfície do torus usando os ângulos alpha e beta, e conectando esses pontos para formar triângulos. Isso resulta num conjunto de triângulos que aproxima a superfície do toro.

Visto que os ângulos αe β variam para cada ponto, é nos permitido calcular todas as coordenadas dos pontos necessários utilizando as seguintes coordenadas cilíndricas:

- x = (outerRadius + innerRadius * cos(β)) * cos(α)
- y = innerRadius * sin(β)
- z = (outerRadius + innerRadius * cos(β)) * sin(α)

Tomando como exemplo o quadrilátero formado pelos pontos 1, 2, 3 e 4, semelhante ao descrito na figura 8. Para calcular as coordenadas dos pontos 2, 3 e 4 em relação ao ponto 1, precisamos determinar os valores dos ângulos correspondentes. Para o ponto 2, em relação ao ponto 1, apenas o ângulo beta é alterado (aumenta 1 unidade relativamente à divisão atual), para o ponto 3, ambos os ângulos são alterados, e para o ponto 4, é alterado o ângulo alpha (aumenta 1 unidade em relação à divisão atual do ângulo). Assim, os ângulos são calculados utilizando as fórmulas:

- $alpha = \frac{2*pi}{slices} * a$, onde $a \in [0, slices[$
- $beta = \frac{2*pi}{stacks} * b$, onde $b \in [0, stacks[$

Por exemplo, considerando os pontos 1 e 3 da figura 8, se para o ponto 1, $\alpha = \frac{2*pi}{slices} * a$ e $\beta = \frac{2*pi}{stacks} * b$, então para o ponto 3, $\alpha = \frac{2*pi}{slices} * (a+1)$ e $\beta = \frac{2*pi}{stacks} * (b+1)$.
Aplicando as coordenadas esféricas com estes valores dos ângulos temos as coordenadas x, y e z de todos os pontos do toro.

3.1.6 Escrever no Ficheiro

Para efetuar a escrita de pontos no ficheiro passado como argumento de entrada ao Generator, foi criada uma função writePoint que recebe um objeto Point e um objeto ofstream como parâmetros. O objetivo desta função é escrever tanto o número de pontos totais como as coordenadas do ponto dado no fluxo de arquivo de saída especificado (writer), separadas por ponto e vírgula e terminadas por um caractere de nova linha.
Na implementação das primitivas, após abrir o ficheiro em modo escrita, a utilização das funções supracitadas permitem que a transcrição de pontos seja realizada cada vez que se calcula um ponto, isto é, os pontos não são armazenados em nenhuma estrutura. Esta metodologia foi utilizada visando poupar memória, fazendo com que o programa seja o mais eficiente possível.

```
54
-1;0;-1
-0.333333;0;-1
-1;0;-0.333333
-0.333333;0;-1
```

Figura 12: Excerto do formato do ficheiro .3d

3.2 Engine

3.2.1 Leitura do ficheiro XML

Uma vez que a informação relativa aos ficheiros que devem ser carregados se encontra num documento XML(passado como argumento ao Engine), utilizou-se a biblioteca TinyXML-2 para realizar o *parsing* desse mesmo documento, com o objetivo de decifrar quais os modelos a desenhar. A função parseXML é responsável por analisar um arquivo XML que contém informações sobre o modelo de uma dada primitiva e todas as componentes necessárias para renderizar a cena. Ele extrai informações como as dimensões da janela, configurações da câmera como posição, direção do olhar, vetor de up e configurações de projeção. Calcula também o raio, ângulos alpha e beta para a câmera. Para além disso, extrai informações sobre os modelos a serem carregados e adiciona-os a um vetor de modelos. Essas informações são todas armazenadas com o auxílio de estruturas.

```
// Structure to store window settings
struct Window {
    float width, height;
};
```

Figura 13: Estrutura de informações da janela

```
// Structure to store camera settings
struct Camera {
    float posX, posY, posZ;
    float lookAtX, lookAtY, lookAtZ;
    float upX, upY, upZ;
    float fov, nearPlane, farPlane;
};
```

Figura 14: Estrutura de informações da câmera

```
// Vector to store all the models read in the xml
vector<string> models;

// Vector to store all the points to be drawn
vector<Point> points;
```

Figura 15: Estrutura de informações de vectores

3.2.2 Leitura dos modelos

Após sabermos quais os modelos que devem ser carregados, procedeu-se à leitura dos mesmos. Para este efeito percorre-se o vetor models anteriormente preenchido e, para cada um dos modelos, é invocada a função loadModel. A função loadModel abre o ficheiro, relativo ao modelo que recebe como argumento, e percorre-o de linha em linha, até alcançar o fim do ficheiro. Dado que cada linha de informação representa um vértice, e que a ordem pela qual esses vértices estão escritos permite desenhar os triângulos necessários à formação correta da primitiva, a função limita-se a armazenar os pontos que vai lendo no vector de pontos.

3.2.3 Desenho das primitivas

Para desenhar as primitivas, cujos vértices foram anteriormente carregados para memória, criou-se a função drawPrimitives. A função drawPrimitives percorre points 3 a 3 e desenha os triângulos necessários à criação da primitiva com o auxílio da função glVertex3f (definindo antes a opção GL TRIANGLES) da biblioteca GLUT. Para uma compreensão mais clara da estrutura das primitivas, utilizando a função glColor3f, os triângulos que as compõem são desenhados em diferentes cores, variando entre rosa e amarelo. A fim de tornar a perceção mais nítida, implementamos a opção de trocar entre faces preenchidas ou linhas delimitadoras em branco.

3.2.4 Câmera

Como já mencionado anteriormente, o ficheiro xml também terá informações sobre a câmera e após estas serem devidamente guardadas, estas serão utilizadas na função gluLookAt que nos permite definir três parâmetros:

1. **camera.posX, camera.posY, camera.posZ**: As coordenadas da posição da câmera no espaço tridimensional. Esta é a posição onde a câmera está localizada.

2. **camera.posX, camera.posY, camera.posZ**: As coordenadas do ponto de referência que a câmera está a focar.

3. **camera.upX, camera.upY, camera.upZ**: As coordenadas do vetor que define a direção "para cima"da câmera. Este vetor especifica a orientação da câmera em relação ao mundo tridimensional.

```
gluLookAt(camera.posX,camera.posY,camera.posZ,
          camera.lookAtX,camera.lookAtY,camera.lookAtZ,
              camera.upX,camera.upY,camera.upZ);
```

Figura 16: Função gluLookAt

Basicamente, a associação da estrutura Câmera e desta função calcula uma matriz de visualização que transforma todos os pontos da cena para a perspetiva correta, de modo que a cena seja gerada conforme os requisitos estabelecidos.

3.2.5 Eixos

Como funcionalidade opcional, o Motor permite utilizar um sistema de eixos tridimensional para facilitar a visualização das primitivas. Para desenhar os eixos foi implementada a função drawAxes que desenha um eixo vermelho, verde e azul, para representar o x, o y e o z, respetivamente.

O desenho dos eixos é feito recorrendo, mais uma vez, às funções glVertex3f (definindo antes a opção GL LINES) e glColor3f. Cada eixo tem uma dimensão de 100 a -100 unidades. Para ativar ou desativar a opção apresentar os eixos, basta clicar na tecla A, que altera o valor da variável axes sucessivamente, para true ou false. Esta funcionalidade é especificada pela função processKeys, que e posteriormente validada através da função glutKeyboardFunc do GLUT.

3.2.6 Modo com linhas

Em vez de visualizar sempre a primitiva com as faces dos triângulos preenchidas, é possível escolher uma opção de desenho em que apenas se visualizam linhas. Isto é conseguido através da utilização da função glPolygonMode com as opções glPolygonMode(GL FRONT AND BACK, GL LINE) para se verem as linhas, ou glPolygonMode(GL FRONT AND BACK, GL FILL) para se verem novamente as faces. Para ativar ou desativar a opção apresentar o modo de linhas, basta clicar na tecla L, que altera o valor da variável lines sucessivamente, para true ou false. Esta funcionalidade é especificada pela função processKeys, que é posteriormente validada através da função glutKeyboardFunc do GLUT.

3.2.7 Movimentação da câmera

Como foi dito anteriormente, a câmera tem a possibilidade de se movimentar graças à utilização da função gluLookAt. No entanto, a posição da câmera baseia-se no valor das variáveis globais alpha, beta e radius. Para mover a câmera 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 câmera da origem, basta clicar em + ou -, respetivamente, aumentando e diminuindo o radius. Esta funcionalidade é especificada pela função processKeys, que é posteriormente validada através da função glutKeyboardFunc do GLUT.

4 Resultados Obtidos

De forma a testar todo o trabalho elaborado, utilizamos os ficheiros disponibilizados na pasta de testes relativamente à fase 1 do projeto. Como é possível observar, todas as cenas estão idênticas ao esperado, o que consideramos que tenha sido um sucesso. Para o teste do toros, criamos um ficheiro XML com os mesmos parâmetros exigidos, sendo a demonstração disponibilizada na figura 16.

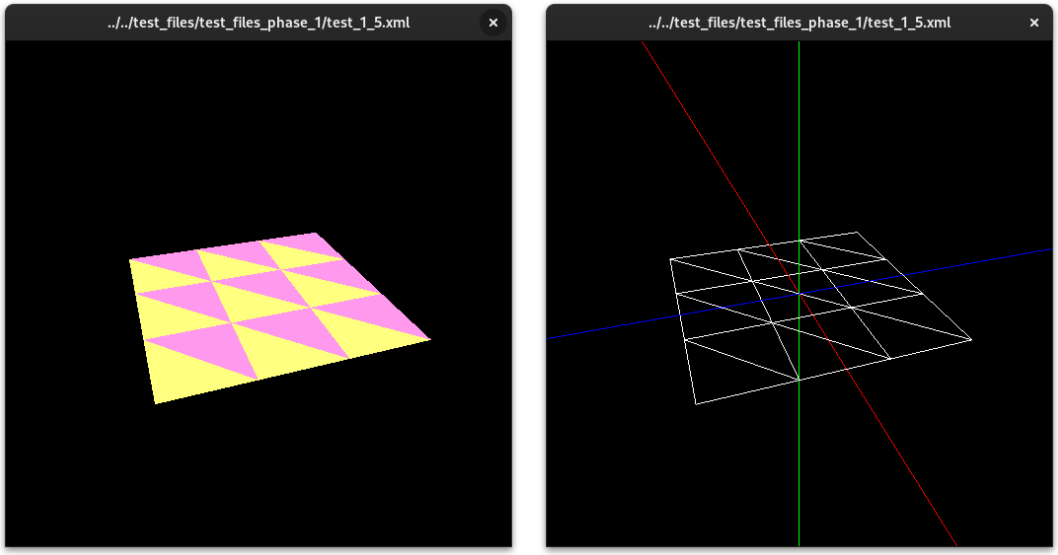


Figura 17: Plano

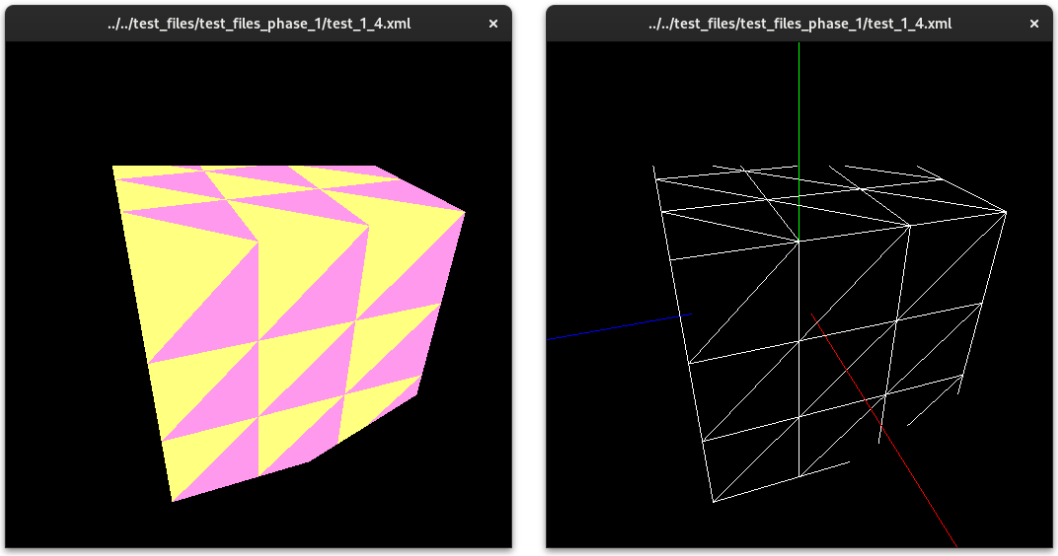


Figura 18: Cubo

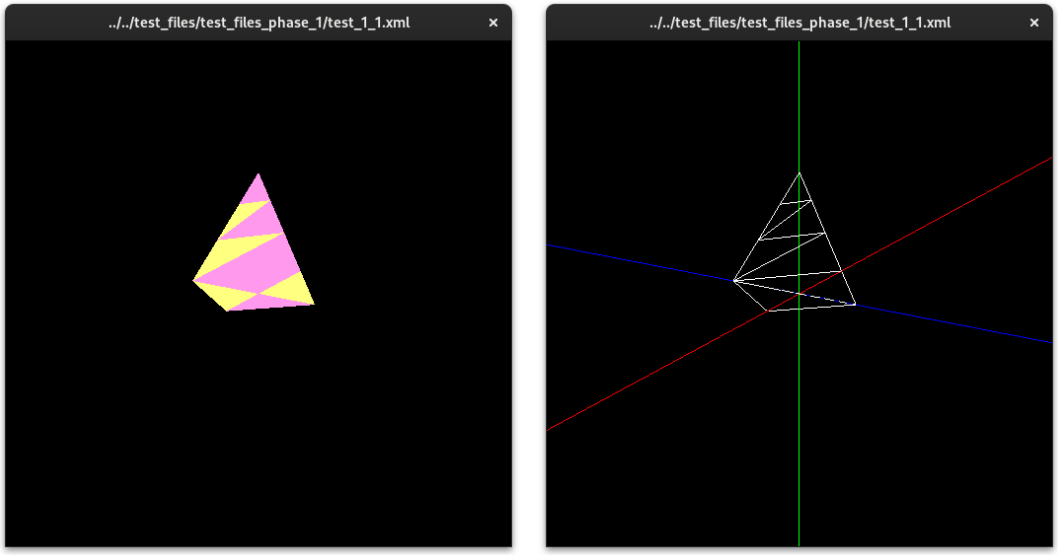


Figura 19: Cone

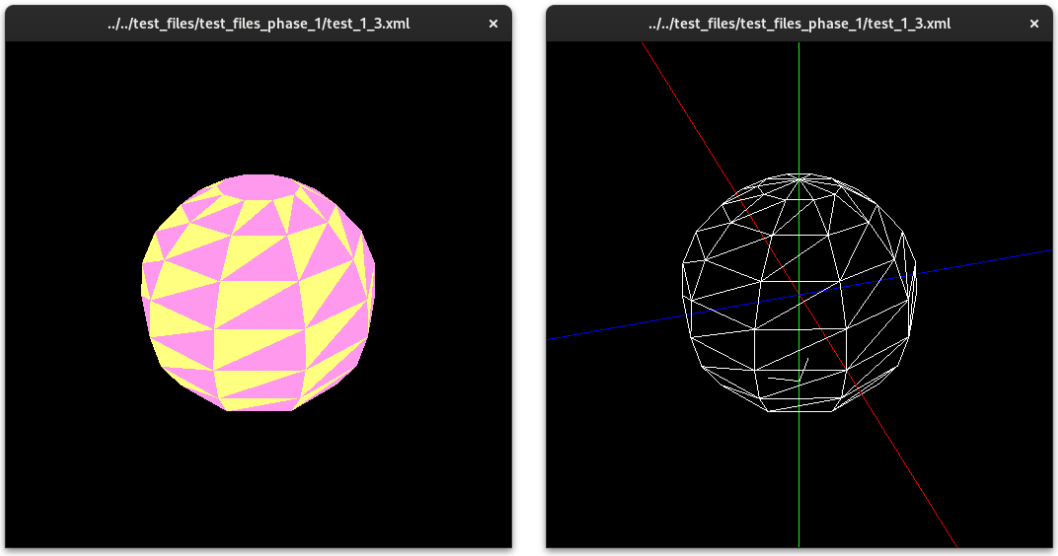


Figura 20: Esfera

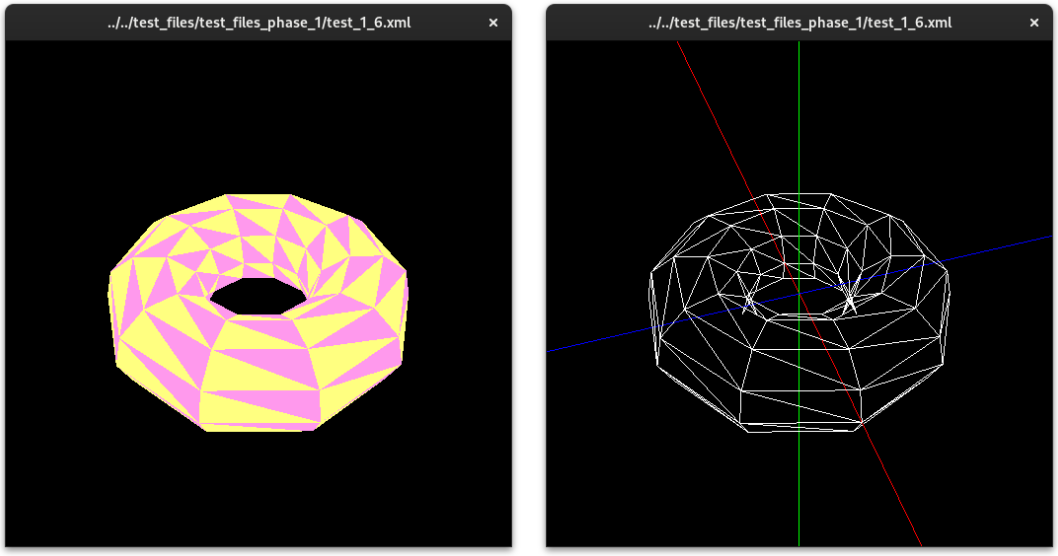


Figura 21: Toro

5 Conclusão

Fazendo uma retrospetiva da fase consideramos que foi um sucesso uma vez que cumprimos todos os requisitos estabelecidos e ainda decidimos implementar uma primitiva adicional, o toro. Esta implementação foi ponderada pensando já no objetivo final do projeto da implementação do sistema solar.

A realização desta primeira fase foi crucial para estabelecer uma base sólida para o restante desenvolvimento do projeto. Agora, com uma compreensão mais profunda das bibliotecas e ferramentas utilizadas, acreditamos que estamos numa posição mais vantajosa para avançar para a próxima fase.

A familiaridade adquirida com as ferramentas e a linguagem de programação durante a fase um simplificará a execução da próxima tarefa. Isso porque consideramos que estamos mais confortáveis com os procedimentos e métodos necessários para manipular geometrias e implementar as transformações desejadas.