Universidade do Minho
Escola de Engenharia

Computação Gráfica

Fase1 – Graphical primitives

Grupo 39:

André Vaz
João Mendes
Laura Rodrigues
Luís Fernandes

March 13, 2022

A93221 A93256 A93169 A88539

Fase1 – Graphical primitives

A. Vaz, J. Mendes, L. Rodrigues, L. Fernandes

Universidade do Minho, Departamento de Informática
R. da Universidade, 4710-057 Braga, Portugal
e-mail: {a93221,a93256,a93169,a88539}@alunos.uminho.pt
https://www.uminho.pt

Table of Contents

1 Introduction

A primeira fase do trabalho prático é constituida pelo desenvolvimento de um gerador de vértices de diferentes primitivas gráficas como plano, caixa, esfera e cone, tendo em consideração os diferentes parâmetros, como a altura, largura, profundidade e, por fim, o número de divisões. Bem como, o desenvolvimento de um motor que lê o ficheiro XML e que contem as configurações da câmera, que por fim resultará na exibição das primitivas gráficas.
Consequentemente, iremos explicar as estratégias aplicadas para o desenvolvimento das primitivas gráficas principais tal como para as primitivas gráficas extras, da mesma maneira que também exibimos as suas representações gráficas.

2 Decisions and Approaches

De seguida passamos a explicar o tratamento para o desenvolvimento das várias primitivas e a sua exibição gráfica.

2.1 Utils

Conjunto de ferramentas usadas tanto no *generator* bem como na *engine*.

Point Definimos a classe Ponto que tem como variaveis um conjunto de três *float* que correspondem às cooordenadas (x,y,z). Para além disso, definimos os seus respetivos *getters* e *setters*.

Patch Nesta classe definimos um *array* dinâmico que contêm um conjunto de pontos. Inclui funções que permitem adicionar pontos a este *array* e obtê-los.

Model Definimos um *array* dinâmico que contêm um conjunto de *patches*. Esta classe inclui funções que permitem adicionar *patches* a este *array* e obtê-las, inclui também funções de ler e escrever num ficheiro. Em conjunto com *engine* foi desenvolvida uma função que permite desenhar o modelo na interface gráfica com o apoio do *openGL*.

2.2 Generator

Aqui elaboramos todas as funções que correspondem as diferentes primitivas desenvolvidas.

Plano Para a construção do plano elaborámos a função *plane* que tem como argumentos a unidade de comprimento (*units*) as divisoes (*divisions*) e o ficheiro .3d (*filename*) - onde armazena os pontos no ficheiro recebido - originando um plano ao longo do eixo xOz.
Para a construção do plano, primeiramente, determinou-se o número de pontos que o constituem cada secção quadrada que, neste caso são 6. Consequentemente determina-se os valores das coordenadas x e z de acordo com as seguintes expressões, de modo a obter um plano centrado na origem do plano xOz:

$$\mathrm{x} = -units/2$$

$$\mathrm{y} = 0$$

$$\mathrm{z} = -units/2$$

Os pontos das várias secções são calculados incrementando a estes a divisão em que se encontram. Usámos a variável *div*, ($units/divisions$), ou seja, o comprimento de cada secção, para incrementar apenas os valores de x e z, uma vez que o y não se altera.

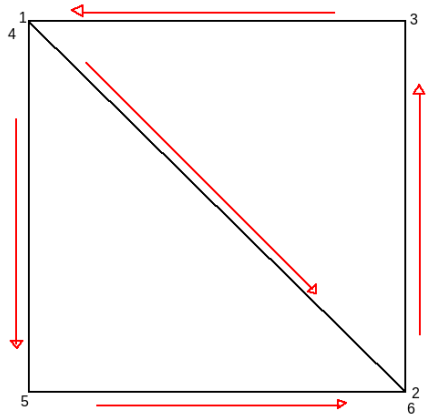

Fig. 1. Ordem dos pontos

De modo a tornar o plano visível visto de cima, foi necessário ordenar os pontos pelo sentido contrário aos ponteiros do relógio. Assim, o primeiro triângulo da secção é definido pelos pontos 1, 2 e 3 e o segundo por 4, 5 e 6.

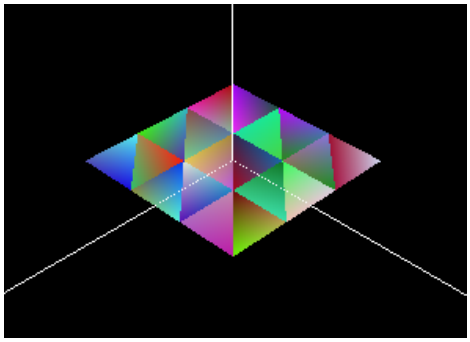

Fig. 2. Plano - plane 1 3 plane.3d

Caixa Para a elaboração da caixa desenvolvemos a função *box*, para a qual nos baseámos no processo utilizado para a elaboração do plano, por isso esta função recebe como argumentos a unidade de comprimento (*units*) as divisões (*grid*) e o ficheiro .3d (*filename*) que armazena os pontos que constituem a caixa desenvolvida.

1. **Definição das Variáveis**

   Estas permitiram calcular as distância das faces aos eixos, facilitando o cálculo das posições em que cada ponto iria estar.
   x = $-units/2$
   y = $-units/2$
   z = $-units/2$

2. **Elaboração dos planos**

   Tal como no plano, os pontos seguintes foram calculados a partir das coordenadas do ponto inicial, aos quais foi sendo incrementada *div*.
   $div = units/grid$ é, assim, a distancia entre cada secção.
   Para cada plano foi repetido o processo.

3. **Resultado**

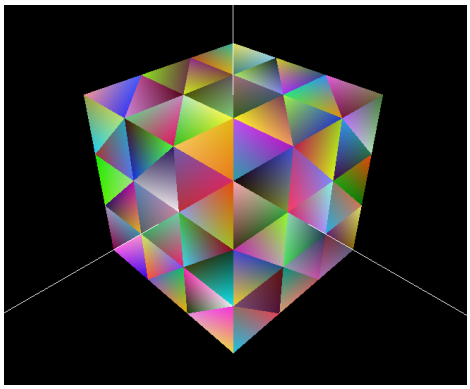

Fig. 3. Caixa - box 2 3 box.3d

Esfera Para a criação da esfera desenvolvemos a função *sphere*, que recebe como argumentos o raio (*radius*), as fatias (*slices*), as camadas (*stacks*) e o ficheiro .3d (*filename*) que armazena os pontos que constituem a esfera elaborada.

1. **Definição das Variáveis**
   A variação da distância entre cada slice e cada stack é calculada, respetivamentea, através das variáveis *deltah* e *deltav*.

   $deltah = 2 * \pi / slices$
   $deltav = \pi / stacks$

   A partir destas foi possível calcular os ângulos que por sua vez nos deram as coordenadas de cada ponto. Através do gráfico abaixo podemos deduzir as equações necessárias.

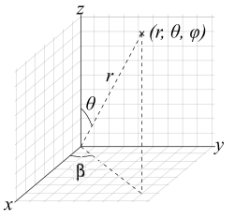

Fig. 4. Coordenadas esféricas

$$x = x_0 + r \sin\theta \cos\beta$$
$$y = y_0 + r \sin\theta \sin\beta$$
$$z = z_0 + r \cos\theta$$

Fig. 5. Equações das coordenadas esféricas

Sabendo que podíamos calcular as coordenadas de cada ponto usando as expressões acima, usámos β 1 e θ 1 para calcular os pontos do lado esquerdo da slice e β 2 e θ 2 para calcular os do lado direito.

$beta1 = i * deltah$
$beta2 = (i + 1) * deltah$
$theta1 = j * deltav$
$theta2 = (j + 1) * deltav$

2. **Elaboração das Slices**
   A iteração foi feita por slices. Para cada uma desenhou-se o triângulo de cima, os das stacks centrais e o de baixo.

3. **Resultado**

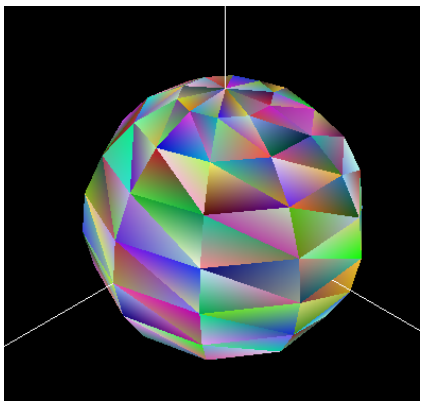

Fig. 6. Esfera - sphere 1 10 10 sphere.3d

Cone Para a realização do cone executámos a função *cone* que tem como argummentos os seguintes parâmetros: o raio (*radius*), a altura (*height*), as fatias (*slices*), as camadas (*stacks*) e, por último, o ficheiro .3d, que armazena os pontos que constituem o cone.

1. **Definição das Variáveis**
   Tal como na esfera, a variação da distância entre cada slice e cada stack é calculada, respetivamentea, através das variáveis *deltah* e *deltav*. A variável *deltaRaio* permitiu calcular a variação do raio de cada stack.

   $deltah = 2 * \pi / slices$
   $deltav = \pi / stacks$
   $deltaRaio = radius / stacks$
   $y = height / 2$

   Graças às variações horizontais (*deltah*) obtivemos os ângulos que nos permitiram calcular as coordenadas x e z, recorrendo às transformações *sin* e *cos*. Para cada uma destas multiplicámos também pelo raio da stack em questão.

   $angle1 = i * deltah$
   $angle2 = (i + 1) * deltah$
   $raioStack1 = radius - j * deltaRaio$
   $raioStack2 = radius - (j + 1) * deltaRaio$

   As coordenadas y foram calculadas recorrendo à altura da stack atual (*alturaS*) e à da stack seguinte (*alturaS2*, as quais dependem da variação vertical, ou seja, a distância entre cada stack).

   $alturaS = -y + j * deltav$
   $alturaS2 = -y + (j + 1) * deltav$

2. **Elaboração dos Stacks**
   No caso do cone, a figura é construída por stacks. Começamos por criar a face de baixo por slices para a stack 0, prosseguindo depois para elaboração das laterais. Nas laterais é aplicado o mesmo sistema de ordenação de pontos usado para o plano (8, com exceção da última stack, que é feita como uma pirâmide (3 pontos para cada slice).

3. **Resultado**

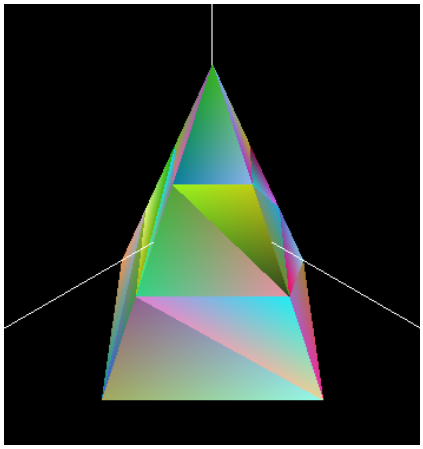

Fig. 7. Cone - cone 1 2 4 3 cone.3d

Tronco Como primitiva extra decidimos elaborar o tronco, para tal desenvolvemos a função *tronco* que tem os seguintes argumentos *radius1*, que corresponde ao raio da circunferência de cima, *radius2*, que corresponde ao raio da circunferência de baixo, a altura (*height*), as fatias (*slices*, as camadas (*stacks* e o ficheiro .3d, que armazena os pontos do tronco. A partir desta função também é possivel formar o cilindro, se o valor dos raios forem os mesmos.

1. **Definição das Variáveis**
   Tal como no cone, a variação da distância entre cada slice e cada stack é calculada, respetivamentea, através das variáveis *deltah* e *deltav*. A variável *deltaRaio* permitiu calcular a variação do raio de cada stack e é o valor absoluto da diferença uma vez que não há restrições a que a face de cima seja maior que a de baixo ou vice-versa.

   $deltah = 2 * \pi / slices$
   $deltav = height / stacks$
   $deltaRaio = |radius1 - radius2| / stacks$
   $y = height / 2$

   Mais uma vez, o cálculo de x e z é feito graças a *sin* e *cos* do *angle1* e *angle2* que recorrem a *deltah*. As coordenadas y foram calculadas recorrendo à altura da stack atual (*alturaS*) e à da stack seguinte (*alturaS2*.

   $angle1 = i * deltah$
   $angle2 = (i + 1) * deltah$
   $alturaS = -y + j * deltav$
   $alturaS2 = -y + (j + 1) * deltav$

Por sua vez o cálculo do raio da stack depende de se o raio 1 é maior ou menos ao 2.

Caso (radius1 < radius2):
$raioStack1 = radius2 - j * deltaRaio$
$raioStack2 = radius2 - (j+1) * deltaRaio$

Caso (radius1 > radius2):
$raioStack1 = radius2 + j * deltaRaio$
$raioStack2 = radius2 + (j+1) * deltaRaio$

2. **Elaboração das Stacks**
   O desenho da figura é feito por stacks. Para a primeira stack (j=0) é desenhada também a base de baixo do tronco. Na última stack (j=stacks-1) é desenhada também a base de cima. Para ambas as bases o desenho é feito à semelhança da base do cone.

3. **Resultado**

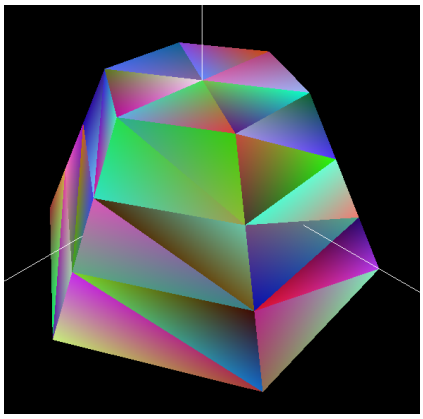

Fig. 8. Tronco - tronco 2 2 2 20 3 tronco.3d

2.3 Engine

Para que fosse possível ler e desenhar triângulos através da informação contida num ficheiro XML, desenvolvemos as seguintes classes.

Camera Conjunto de variáveis que correspondem aos constituintes da câmera, sendo estas *position*, *lookAt*, *up*, *fov*, *near* e *far*. Para além disso foram elaborados os seus respetivos *getters* e *setters*.

World Constituido por uma *camera* e um *array* dinâmico de modelos. Nesta classe, para além da elaboração dos seus *getters* e *setters* e da função de adicionar um *model*, desenvolvemos funções que permitem ler ficheiros *XML* carregando a informação neles contida para a classe.
Utilizando a camera e os modelos carregados conseguimos então dar *render* da janela.

3 Testes

De seguida expomos alguns testes de acordo com as medidas dadas no enunciado (diferentes perspetivas, aproximações e combinações):

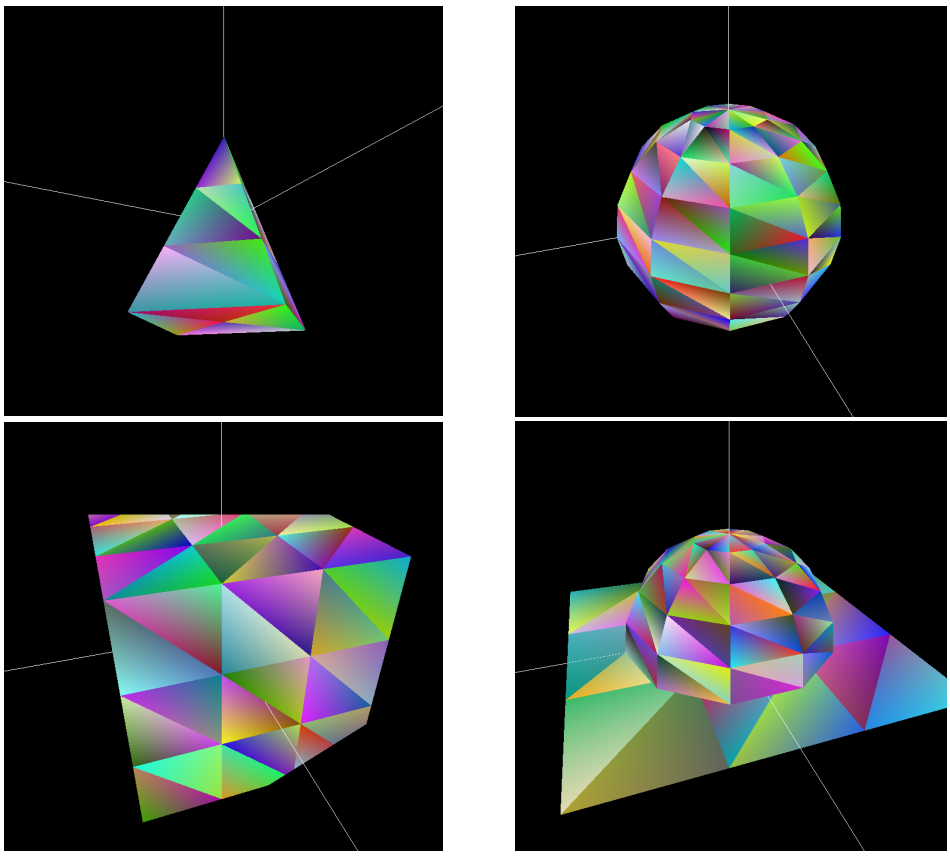

Fig. 9. Testes exemplificativos

4 Conclusion

O projeto teve como intuito a elaboração de um gerador de pontos que constituíssem as diferentes primitivas, para de seguida elaborá-las e exibi-las graficamente. Numa primeira fase sentimos um pouco de dificuldades relativamente ao cálculo matemático dos pontos de algumas primitivas em causa.

Contudo, houve o cuidado de cumprir todos os parâmetros pedidos bem como desenvolver alguns extras, como por exemplo, implementação de primitivas extras, assim como a implementação da cores aleatórias para colorir os triangulos que constituem os planos, que serviu para uma melhor visualização dos mesmos e uma melhor interpretação das figuras.

Concluindo, consideramos que obtivemos um balanço positivo, uma vez que apesar das dificuldades sentidas, cumprimos todos os requisitos.