Universidade do Minho

Computação Gráfica

Relatório Projeto Prático – Parte I

Mestrado Integrado em Engenharia Informática

Ano Letivo 2016/2017

Docente: António José Borba Ramires Fernandes

Elementos do Grupo:

Daniel Teixeira Militão – A74557

Hugo Alves Carvalho – A74219

João Ismael Barros Dos Reis – A75372

Luís Miguel Da Cunha Lima – A74260

Índice

1. Introdução

O presente relatório descreve a primeira etapa do projeto prático da unidade curricular de Computação Gráfica. Nesta fase, foi solicitada a realização de duas aplicações: *generator* e *engine.*

Através de uma *graphical primitive* e as suas características (por exemplo, o lado de um plano), o *generator* é capaz de gerar um ficheiro contendo os vértices necessários para formar a figura pretendida.

O *engine* é responsável por ler um ficheiro XML (que contém uma referência para ficheiros gerados pelo *generator)* e desenhar esses modelos no ecrã.

Neste relatório vamos explicar, através de equações, figuras e algoritmos em pseudocódigo, o nosso raciocínio para a realização das duas aplicações mencionadas.

2. ***Generator***

2.1. Considerações sobre os ficheiros criados

Cada função recebe como parâmetro uma *string* com o nome pretendido para o ficheiro que guarda os pontos que constituem os triângulos necessários para cada figura.

Deste modo, a primeira linha de cada um desses ficheiros contém o número total de triângulos que formam a figura pretendida. Para além disso, cada uma das restantes linhas corresponde a um ponto (com as três coordenadas X, Y e Z). Uma vez que um triângulo é constituído por três pontos, a partir da segunda linha do ficheiro de texto, cada conjunto de três pontos corresponde a um triângulo.

2.2. Plane

A função *makePlane* recebe como parâmetros um *float* que representa a largura de cada lado do plano (*width*).

Com base no comprimento do lado, são gerados os quatro pontos que formam os vértices do quadrado. Com estes quatros pontos é possível construir os dois triângulos que constituem o quadrado através do seguinte algoritmo:

```
1. Colocar na variável side o valor de width/2


2. Criar o primeiro triângulo com os pontos:
            P1-> (side, 0, -side)
            P2-> (-side, 0, -side)
            P3-> (-side, 0, side)


3. Criar o segundo triângulo com os pontos:
            P1 -> (side, 0, -side)
            P3 -> (-side, 0, side)
            P4 -> (side, 0, side)


4.  Fim do algoritmo
```

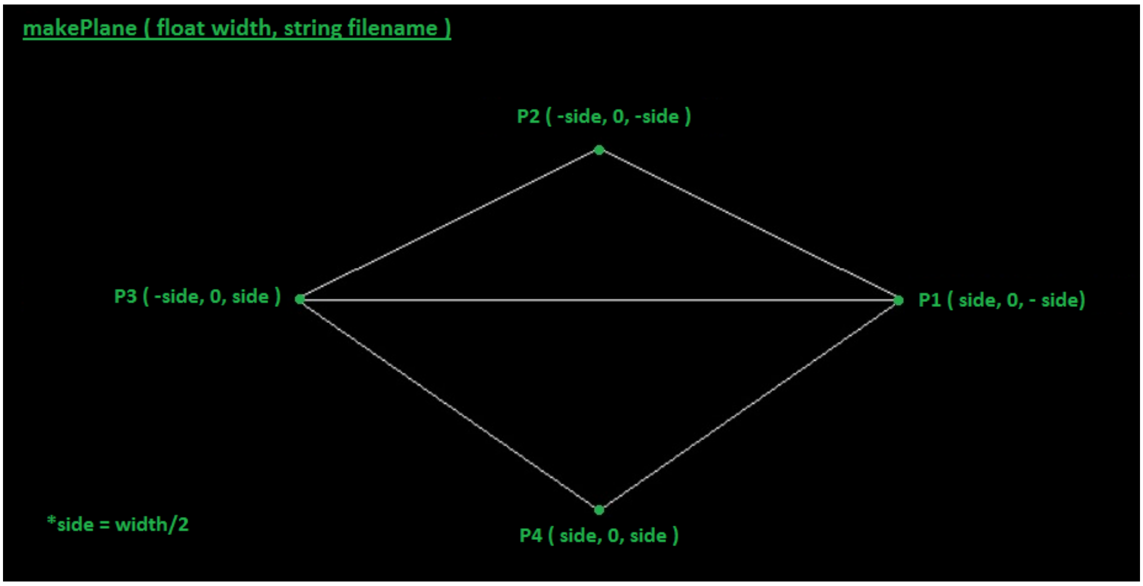


Figura 1 – Vértices do plano

2.3. Box

A função *makeBox* recebe como parâmetros três *floats* que representam as dimensões da caixa segundo os eixos X, Y e Z, um inteiro que representa o número de divisões que cada fase tem.

Uma vez que o número de pontos a gerar depende do número de divisões, foi necessário desenvolver um algoritmo que fosse válido para qualquer valor.

```
1. Colocar na variável py o valor de y/2

       1.1. Se py é igual a -y/2, saltar para o passo 4

2. Colocar na variável px o valor de -x/2

       2.1. Se px é igual a x/2, saltar para o passo 3

       2.2. Criar o primeiro triângulo com os pontos:
            P1 -> (px, py, z/2)
            P2 -> (px, py – (y/numberOfDivisions), z/2)
            P3 -> (px + (x/numberOfDivisions), py, z/2)

       2.3. Criar o segundo triângulo com os pontos:
            P2 -> (px, py – (y/numberOfDivisions), z/2)
            P4 -> (px + (x/numberOfDivisions), py – (y/numberOfDivisions), z/2)
            P3 -> (px + (x/numberOfDivisions), py, z/2)

       2.4. px toma o valor de px + (x/numberOfDivisions)

       2.5. voltar a 2.1

3. py toma o valor de py – (y/numberOfDivisions)

4. Fim do algoritmo
```

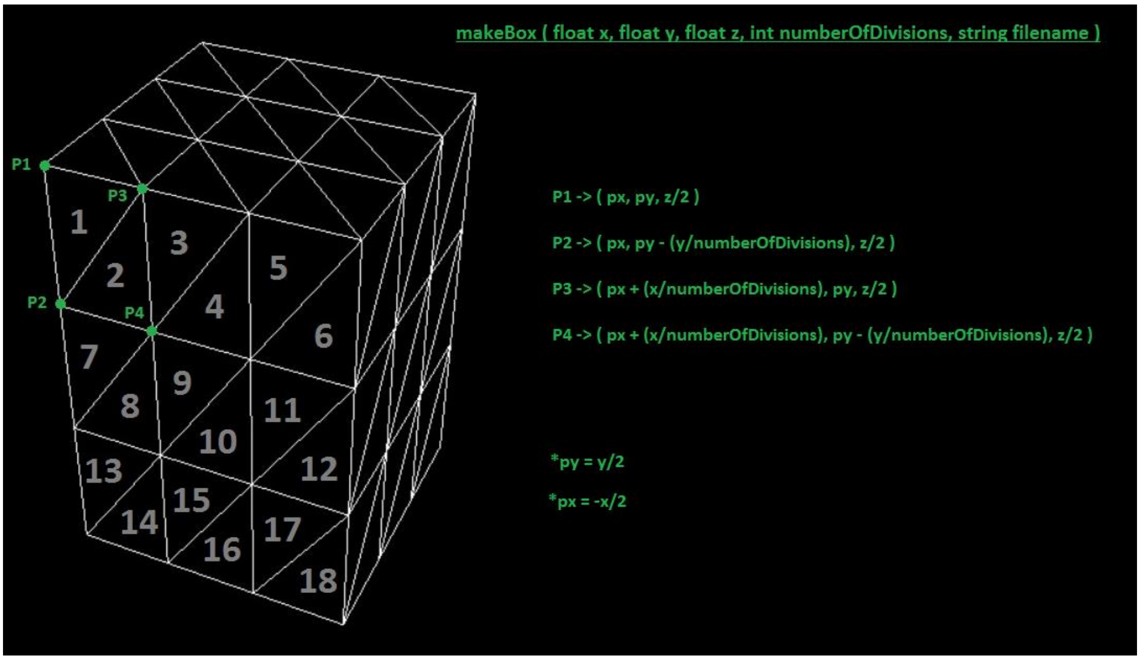


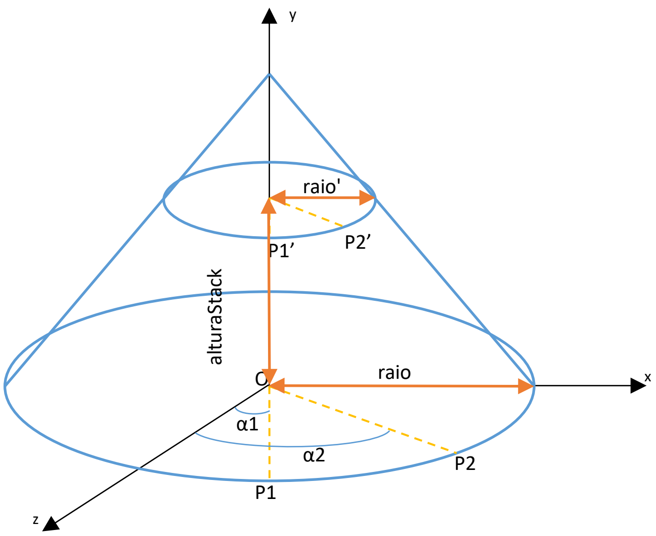
Figura 2 – Vértices da caixa

De modo a explicar o nosso raciocínio no desenvolvimento do algoritmo, podemos verificar na figura 2 a ordem pela qual os triângulos são construídos para a face com o valor de Z fixo e positivo, e com X e Y variáveis.

Para as restantes faces o raciocínio é semelhante, apenas alterando o valor da varíavel fixa e consequentemente, os valores das restantes duas variáveis que vão sendo alteradas ao longo do algoritmo.

2.4. Cone

Para conseguir gerar um cone foi necessário primeiro saber como é que se iriam calcular os diversos pontos do cone. A figura 3 mostra como se calculam pontos da superfície do cone e a figura 4 mostra como é que se calcula o raio e a altura do cone numa determinada *stack*. Com este conhecimento foi-nos possível desenvolver o seguinte algoritmo para cálculo dos vértices.



$$\alpha 1 = numeroSliceAtual * \left(\frac{2\pi}{numeroSlices}\right)$$

$$\alpha 2 = (numeroSliceAtual + 1) * \left(\frac{2\pi}{numeroSlices}\right)$$

Nota: *slices* começam a ser contadas a partir do zero

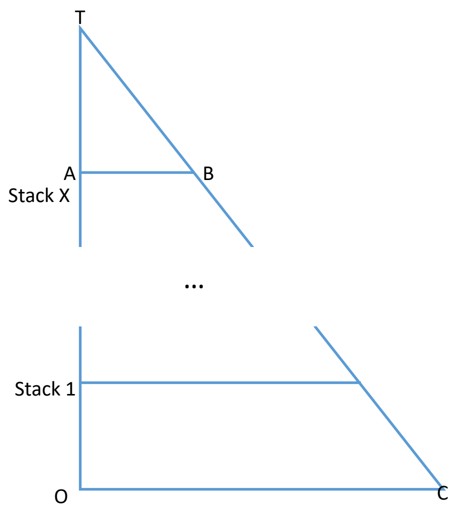
$$P1 \rightarrow (raio * \sin(\alpha 1), 0, raio * \cos(\alpha 1))$$

$$P2 \rightarrow (raio * \sin(\alpha 2), 0, raio * \cos(\alpha 2))$$

$$P1' \rightarrow (raio' * \sin(\alpha 1), alturaStack, raio' * \cos(\alpha 1))$$

$$P2' \rightarrow (raio' * \sin(\alpha 2), alturaStack, raio' * \cos(\alpha 2))$$

Figura 3 – Como calcular as coordenadas de um ponto na superfície do cone



$$\frac{TA}{TO} = \frac{AB}{OC} \Leftrightarrow$$

$$\Leftrightarrow \frac{Altura - AlturaStack}{Altura} = \frac{RaioStack}{Raio} \Leftrightarrow$$

$$\Leftrightarrow RaioStack = \frac{(Altura - AlturaStack) * Raio}{Altura}$$

$$AlturaStack = \frac{Altura}{stacks} * NumeroStack$$

Figura 4 - Como calcular o raio e a altura de uma stack

Cálculo dos vértices:

1. Iterar sobre o número de *slices*
    a. Calcular o ângulo alfa para a *slice* atual
    b. Calcular o ângulo alfa para a próxima *slice*
    c. Calcular os três pontos do triângulo da base que está entre estas duas *slices*
    d. Guardar altura e raio atual para serem usados e substituídos no próximo ciclo
    e. Iterar sobre o número de *stacks* exceto a última (a ultima *stack* é apenas um triângulo, por isso á calculada de maneira diferente)
        i. Calcular a altura da *stack*
        ii. Calcular o raio da *stack*
        iii. Calcular os dois triângulos da *stack* (figura Z)
        iv. Guardar a altura e raio desta *stack*
    f. Calcular o triângulo da ponta do cone (formado pelos dois pontos da stack anterior e a ponta do cone de coordenadas (0, altura, 0))

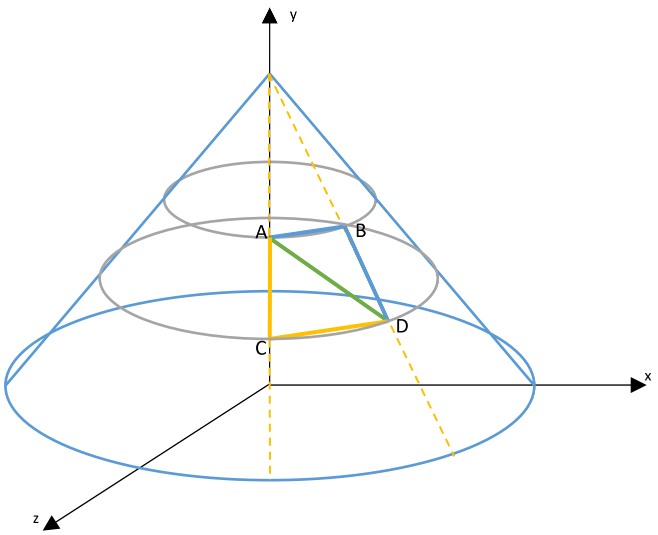


Figura 5 - Triângulos que são calculados em cada iteração do ciclo das stacks ((A,B,C) e (A, B, D))

2.5. *Sphere*

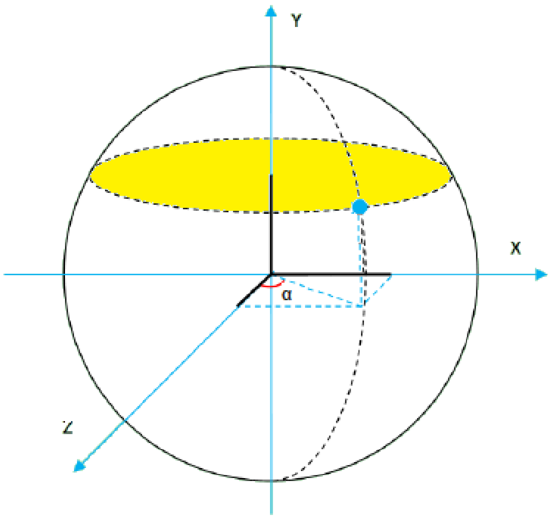


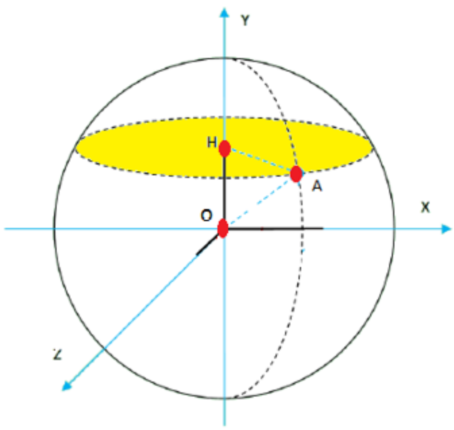
Figura 6 - Gerar Esfera

Com o intuito de gerar os vértices necessários à construção da esfera através do raio, número de *slices* e número de *stacks*, utilizou-se as coordenadas esféricas.

Para esta construção é então necessário um ângulo que varia segundo o eixo Y (representado por **α** na figura acima apresentada).

O número de slices é importante no cálculo dos progressivos novos valores de **α** nas iterações.

α = SliceAtual * ((2 * π) / nº Total de slices);



(Raio da stack em questão) $\overline{HA}$ = sqrt($\overline{OA}$^2 – $\overline{OH}$^2)

Figura 7 – Esfera

É também importante o calcular a altura das várias stacks, bem como o seu raio (ver figura a acima), de modo a que seja possível formar os triângulos que vão compor a esfera. Com base nestes dados são calculados os diversos vértices da esfera:

- x = sin(α) * raio_StackAtual;
- y = altura_StackAtual;
- z = cos(α) * raio_StackAtual;

Seguidamente, resolvemos partir a esfera em três partes: superior, parte do meio e inferior.

Tanto a parte superior como a inferior da esfera são compostas apenas por triângulos, logo foi necessário fazer uma seleção sequencial correta dos vértices. Na figura seguinte apresentamos o que consideramos de parte superior e inferior, bem como a forma correta da escolha dos vértices.

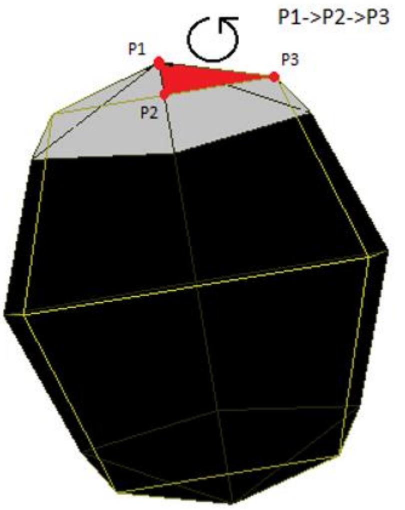


Figura 8 - Parte Superior da Esfera

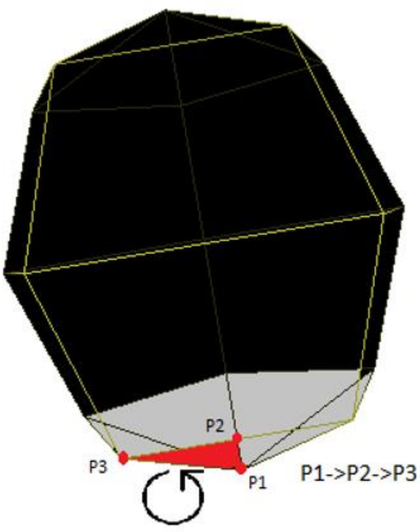


Figura 9 - Parte Inferior da Esfera

Por outro lado, a parte do meio da esfera é formada por quadriláteros, que serão divididos por dois triângulos. O raciocínio é semelhante ao anterior, no entanto são gerados vértices para formar os dois triângulos em cada iteração como podemos ver na figura seguinte.

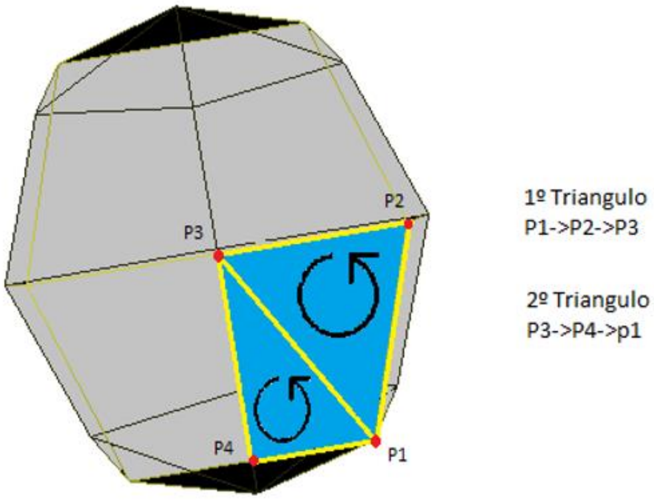


Figura 10 - Parte do Meio da Esfera

De modo a traduzir melhor o nosso raciocínio e facilitar o entendimento do código produzido, apresentamos exemplos dos algoritmos utilizados.

```
// parte superior da esfera

1.Calcular altura e raio da stack atual

2.Colocar na variável "sl" (slice atual) valor 0 e até que "sl" seja menor ou
igual que nº total de slices fazer:

      2.1 Criar triangulo com os pontos

         P1 -> (0, raio da esfera, 0)

         P2 -> (sin(α)*raio da esfera, altura da stack, cos(α)*raio da esfera)

         P3 -> (sin(α)*raio da esfera, altura da stack, cos(α)*raio da esfera)

         com α = (sl + 1) * ((2 * pi) / nº total slices);

      2.2Incrementar "sl"
```

A parte inferior da esfera segue o mesmo raciocínio do anterior apenas variando no calculo da altura, raio da stack e α.

```
//parte do meio da esfera superior

1.Colocar valor 1 na variável "st" (Stack Atual) e enquanto "st" menor que
numero total de stack / 2 fazer:

    1.1. Calcular altura e raio da Stack Atual

    1.2. Colocar valor 0 na variável "st" e enquanto esta for menor que
numero total de slices fazer:

        1.2.1. Criar pontos

        //triangulo inferior

        P1 -> (ponto Atual)

        P2 -> (ponto na mesma slice e stack abaixo)

        P3 -> (ponto a seguir ao atual e stack abaixo)

        // triangulo superior

        P3 -> (ponto a seguir ao atual e stack abaixo)

        P4 -> (ponto a seguir ao atual)

        P1 -> (Ponto atual)
```

A parte do meio da esfera inferior segue o mesmo raciocínio que os exemplos anteriores apenas variando no calculo da altura e raio da stack como o α.

2.6. *Cylinder*

O algoritmo utilizado para o cilindro é muito semelhante ao do cone. Apenas muda ligeiramente a maneira de calcular as *stacks,* dado que o raio se mantém igual ao da base e a última stack tem dois triângulos como as restantes e não um. No topo do cilindro é repetido o mesmo processo que na base, apenas com o *y* igual à altura.

<u>Cálculo dos vértices:</u>

1. Iterar sobre o número de *slices*
    a. Calcular o ângulo alfa para a *slice* atual
    b. Calcular o ângulo alfa para a próxima *slice*
    c. Calcular os três pontos do triângulo da base que está entre estas duas *slices*, tal como foi calculado para o cone (Figura X)
    d. Guardar a altura atual
    e. Iterar sobre o número de *stacks*
        i. Calcular a altura da *stack*
        ii. Calcular os dois triângulos da *stack*
        iii. Guardar a altura desta *stack*
    f. Calcular os pontos do triângulo do topo do cilindro da mesma forma que se calcula os da base, apenas com y igual à altura

2.7. Pyramid

A função *makePyramid* recebe como parâmetros três *floats* que representam o comprimento, altura e largura da pirâmide.

Com base nas dimensões referidas, e uma vez que se trata de uma pirâmide quadrangular, geramos os cinco pontos necessários para formar os seis triângulos que constituem a pirâmide.

```
1. Criar os triângulos para os lados da pirâmide
             1.1. Criar o primeiro triângulo com os pontos:
                    P1-> (0, height, 0)
                    P2-> (-length, 0, width)
                    P3-> (length, 0, width)
             1.2. Criar o segundo triângulo com os pontos:
                    P1 -> (0, height, 0)
                    P3 -> (length, 0, width)
                    P4 -> (length, 0, -width)
             1.3. Criar o terceiro triângulo com os pontos:
                    P1 -> (0, height, 0)
                    P4 -> (length, 0, -width)
                    P5 -> (-length, 0, -width)
             1.4. criar o quarto triângulo com os pontos:
                    P1 -> (0, height, 0)
                    P5 -> (-length, 0, -width)
                    P2 -> (-length, 0, width)
2. Criar os triângulos para a base da pirâmide
             2.1. Criar o quinto triângulo com os pontos:
                    P5 -> (-length, 0, -width)
                    P3 -> (length, 0, width)
                    P2 -> (-length, 0, width)
             2.2. Criar o sexto triângulo com os pontos:
                    P5 -> (-length, 0, -width)
                    P4 -> (length, 0, -width)
                    P3 -> (length, 0, width)
3.  Fim do algoritmo
```

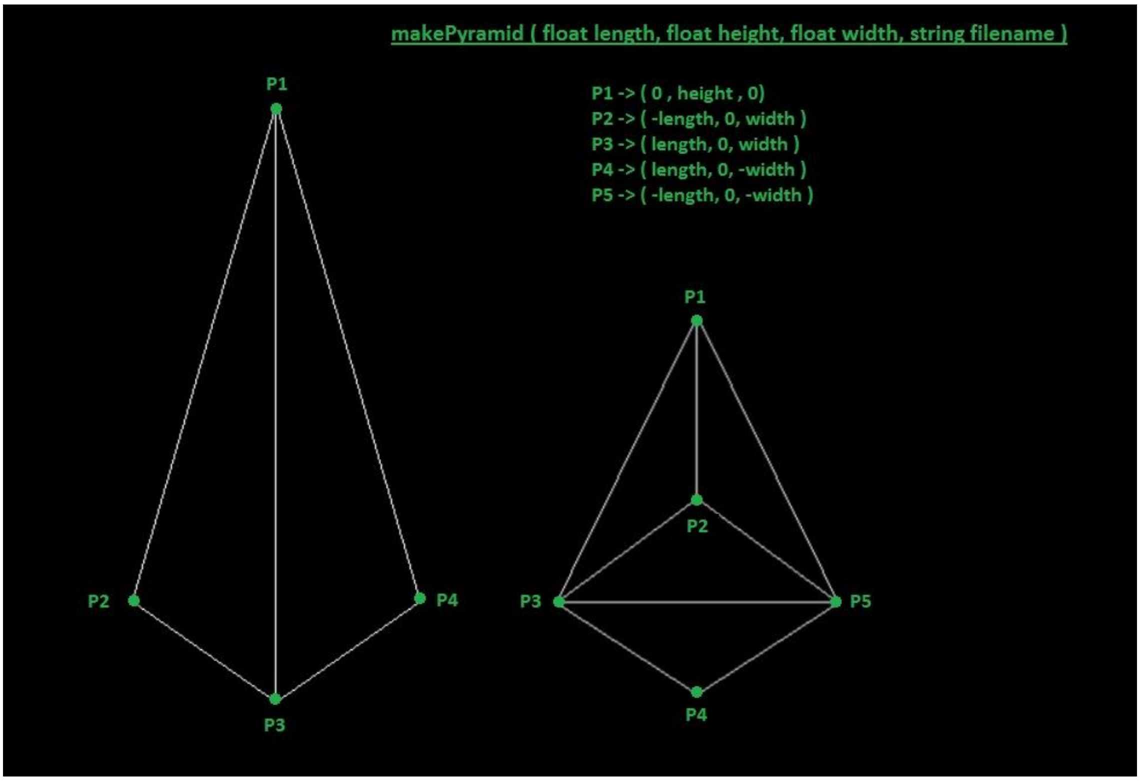


Figura 11 – Vértices da pirâmide

O algoritmo começa por representar os lados da pirâmide, tal como podemos visualizar na primeira imagem da figura 11. Na segunda imagem representada, a pirâmide sofreu uma translação e uma rotação à direita para se poder perceber como estão formados os triângulos que suportam a base da pirâmide.

3. Engine

De modo representar as figuras de uma determinada cena, usou-se XML para referenciar os ficheiros “.3d” construídos pelo gerador. Esses ficheiros contêm várias coordenadas que dão resultado a vários triângulos, resultando em figuras.

```
<scene>
     <model file = "cylinder.3d"/>
     <model file = "box.3d"/>
     <model file = "sphere.3d"/>
     <model file = "plane.3d"/>
     <model file = "pyramid.3d"/>
     <model file = "cone.3d"/>
```

Figura 12 - Exemplo "scene.xml"

Na aplicação criada, existem duas classes:

Coordenate
Contêm um float com os pontos x, y e z.

Figure
Contêm o número de triângulos e um vetor com várias Coordenate.

Assim como um vetor figure que contêm várias Figure.

3.1. Leitura do Ficheiro XML

Assim como sugerido, usou-se a API do tinyXML2 para fazer parse ao documento XML que contem as figuras.

A função getFigures(), que tem como objetivo ler o ficheiro XML e colocar as coordenadas de todos os pontos em memória, divide-se em duas fases:

- <u>Leitura do Ficheiro XML:</u>

    1. Cria vetor onde serão armazenadas as figuras a carregar, vector<string> figuresToLoad;
    2. Abre ficheiro “scene.xml”;
        a. Se não obtiver sucesso, termina execução.
    3. Procura primeiro elemento, “scene”;

a. Se não existir, termina execução.

4. Itera sobre todas as linhas com elemento “model”:
    a. Cria nova string com atributo “file” (que contém o nome das figuras a carregar);
    b. Insere string no vetor `figuresToLoad`.

- Leitura das coordenadas da figura:

1. Itera sobre o vetor figuresToLoad:
    a. Cria objeto `Figure newFig;`
    b. Abre o ficheiro com o nome da string;
    c. Lê do ficheiro o número de triângulos e coloca em `newFig;`
    d. Enquanto ficheiro não chega ao fim:
        i. Cria objeto `Coordenate newC`;
        ii. Coloca coordenadas x,y e z em `newC;`
        iii. Coloca `newC` em `newFig`
    e. Coloca `newFig` em `figures.`

No fim desta função obtemos todas as figuras em memória.

3.2. Representação da figura

Após todas as figuras estarem em memória, falta agora representa-las.

1. Iterar sobre o vetor `figures ;`
    a. Iniciar desenho com `glBegin(GL_TRIANGLES);`
    b. Iterar sobre o vetor que contem os pontos da figura;
        i. Representar o ponto com recurso a `glVertex3f(it->x, it->y, it->z);`
    c. Terminar desenho com `glEnd();`

A cada 3 pontos está também configurado para alternar entre duas cores.

3.3. *Debug*

De modo a ajudar no *Debug* das figuras foi programado uma câmara para melhor visualização e deteção de erros. Com recurso às teclas W, A, S e D é possível mover a figura para cima, esquerda, baixo ou direita, respetivamente. É possível ainda girar usando as setas cima, baixo, direita e esquerda.

Finalmente, foi também incluída a possibilidade de ver as figuras por linhas, pontos ou preenchida usando as teclas Z, X e C, respetivamente.