Universidade do Minho
Mestrado Integrado em Engenharia Informática

Computação Gráfica

Practical Assignment CG - 2019/20
Phase 1: Graphical primitives

Março 2020


Angélica Freitas
(A83761)


Joana Afonso Gomes
(A84912)


Rodrigo Pimentel
(A83765)

Contents

1 Introdução

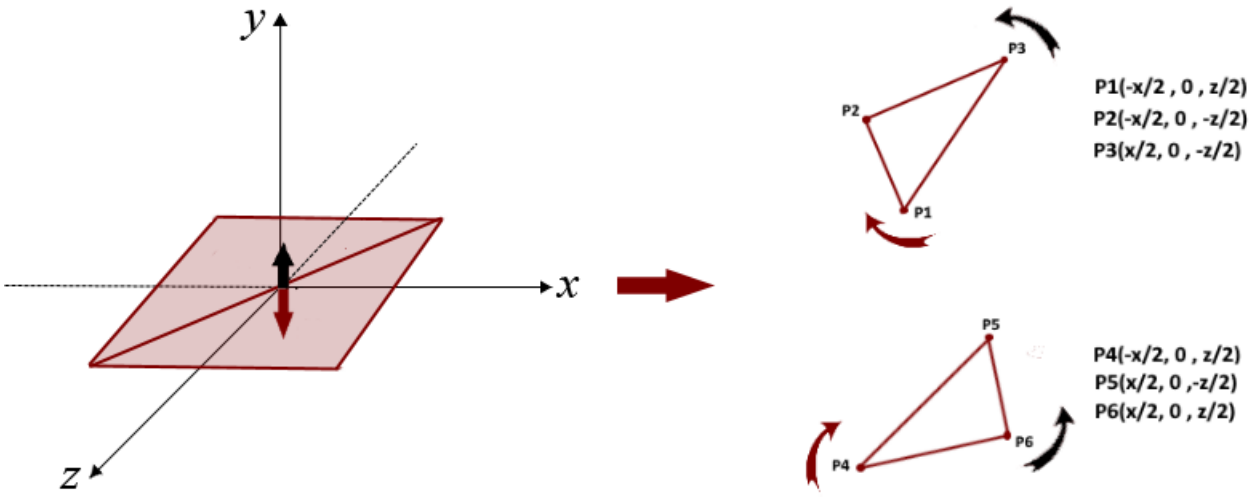
O proposto nesta primeira fase do trabalho prático foi o desenvolvimento de alguns pequenos gráficos 3D. Tal projeto é constituído por duas partes, a primeira baseando-se num gerador de vértices de algumas primitivas gráficas, e a segunda na criação de um mecanismo que permita ler um ficheiro de configuração escrito e XML.

No presente relatório incluímos a apresentação pormenorizada das várias partes do trabalho, ajudando a compreender o processo de elaboração, com base em equações, gráficos, esquemas e figuras.

2 Notas Importantes

Antes de começar a correr código, é de extrema importância ler e acompanhar as instruções para melhor aproveitamento do trabalho (*README*).

3 Plano



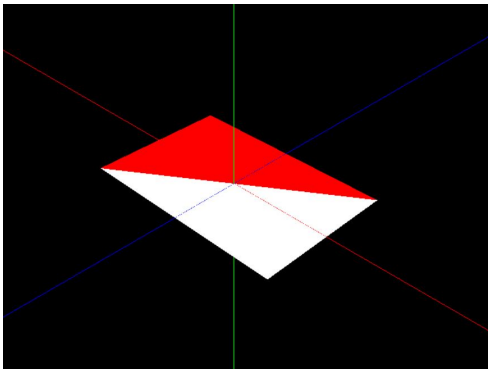
Para obtermos um plano que seja visível em todos os ângulos, temos de implementar a regra da mão direita para os dois sentidos do eixo y (positivo e negativo), como se pode ver pelas setas vermelhas e pretas no referencial.

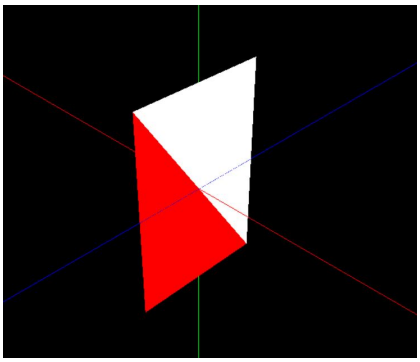
Cada direção (representadas pelas setas pretas e pelas setas vermelhas) será constituída por 2 triângulos, 4 no total. Comecemos pela seta preta. Pela regra da mão direita é perceptível que uma das possíveis ordens dos pontos é a que está representada mais acima (1º Triângulo: P1, P2, P3 e 2º Triângulo: P4, P5, P6). Dividimos por 2 de modo a centrar o plano na origem (0,0,0).

Ora, observando as setas pretas a lado direito da figura de cima, podemos concluir que a ordem que a direção a preto deve tomar é a inversa da direção a vermelho. Isto é, o primeiro triângulo terá ordem P3, P2 e P1 e o segundo triângulo a ordem P6, P5 e P4, visto que são os mesmos pontos vistos anteriormente.

Também, pelas representações dos triângulos acima, é possível observar que o ponto P4 é igual ao ponto P1 e o ponto P5 igual ao P3, decidimos remover os pontos 4 e 5 e reutilizar os pontos 1 e 3.

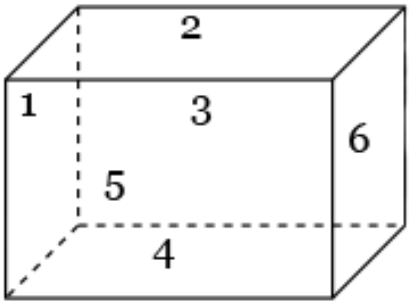
Exemplo de um plano com x= 3 e y=2:





4 *Box*

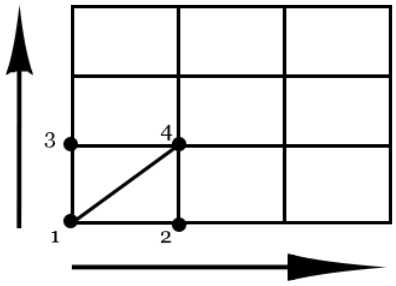
Uma caixa pode ser representada da seguida maneira:



Em que:

- 1 representa a face lateral esquerda;
- 2 representa a face superior;
- 3 representa a face traseira;
- 4 representa a face inferior;
- 5 representa a face frontal;
- 6 representa a face lateral direita.

Inicialmente, de modo a determinar os pontos que seriam necessários gerar, dividimos o processo nas faces da caixa. Para tal e como é possível dividir a caixa, cada face terá, por exemplo, o seguinte aspeto:



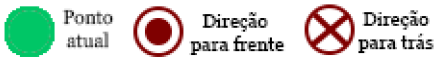
Dependendo da face, os valores de x, y e z vão aumentando, sendo que um destes mantém-se igual. Percorrendo as faces como uma matriz, temos que o ponto 1 é o ponto atual. Como determinar os outros? Como há divisões, temos de calcular — para todos os eixos:

$$xn = x/(n+1)$$
$$yn = y/(n+1)$$
$$zn = z/(n+1)$$

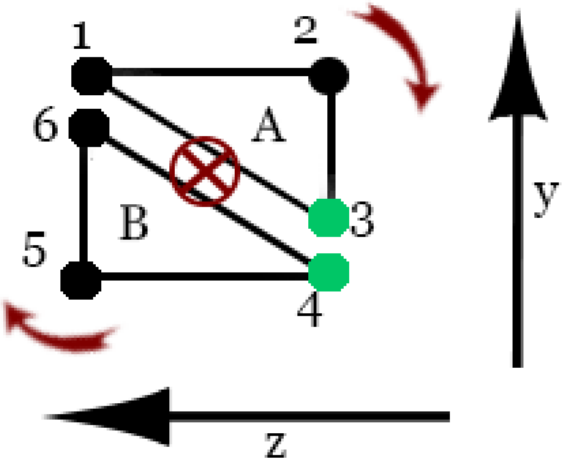
Assim, se o eixo em baixo é x e o do lado é z o ponto 2 representa-se da seguinte maneira (x+xn,y,z). O ponto 3 por (x,y,z+zn) e o ponto 4 por (x+xn,y,z+zn).

Como dito anteriormente, apenas 2 valores se vão alterar. Isto significa que vamos ter de percorrer cada face como uma matriz em que os índices da mesma representam as variáveis que mudam de valor.

De forma a ajudar o processo desenhamos a direção, tendo em conta a regra da mão direita queremos que a direção "aponte para fora" para ser visível em qualquer ângulo, os pontos necessários para representar e as duas variáveis que mudam de valores para cada uma das faces, em que:



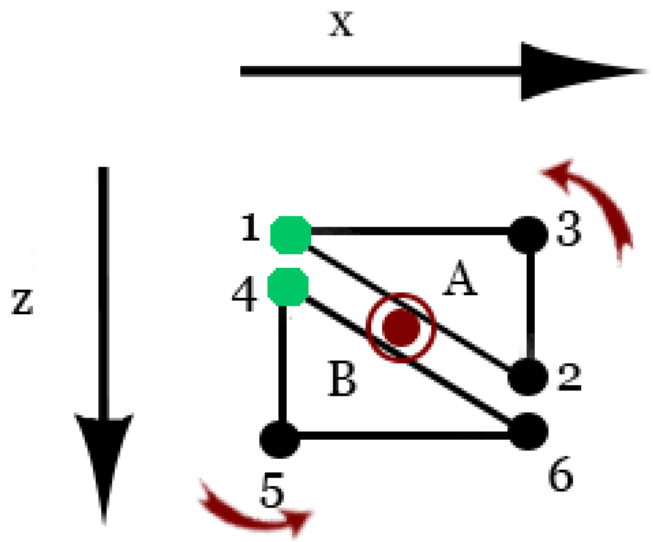
4.1 Face 1



Como estamos a puxar a caixa para a origem (0,0,0), nesta face apenas os valores de y e z vão variar (y toma o valor de 0). Então, vamos iterar em relação ao y e z, sendo que as variáveis de iteração são j e k para y e z, respetivamente. Como o ponto 3 e o ponto 4 representam o ponto atual (0,j,k) de iteração, deduzimos os pontos:

$$Ponto\ 1 = (0,\ j + yn,\ k + zn)$$
$$Ponto\ 2 = (0,\ j + yn,\ k)$$
$$Ponto\ 5 = (0,\ j,\ k + zn)$$
$$Ponto\ 6 = (0,\ j + yn,\ k + zn)$$

4.2 Face 2

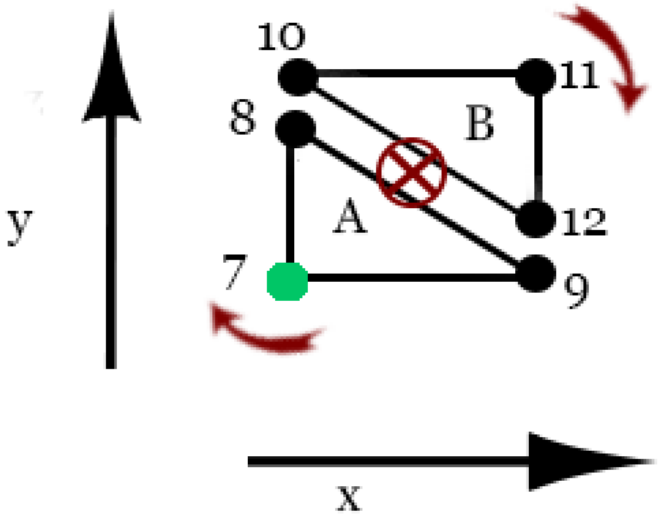


,

Como podemos ver pela figura acima, os valores que vão mudar são o x e o z, i e k respetivamente, o valor de y é o valor dado. Assim com os pontos atuais (i,y,k) obtemos:

$$Ponto\ 2 = (i + xn,\ y,\ k + zn)$$
$$Ponto\ 3 = (i + xn,\ y,\ k)$$
$$Ponto\ 5 = (i,\ y,\ k + zn)$$
$$Ponto\ 6 = (i + xn,\ y,\ k + zn)$$

4.3 Face 3



Tanto nesta face, como na face 5 os índices pelos quais vamos iterar serão o j, que representa o y, e o x que é representado pelo i. Também, nesta face o valor de z para todos os pontos é 0. Deste modo, através do ponto de iteração, que neste caso é o ponto 7 (i,j,0), deduzimos os seguintes pontos:

$$Ponto\ 8 = (i,\ j + yn,\ 0)$$

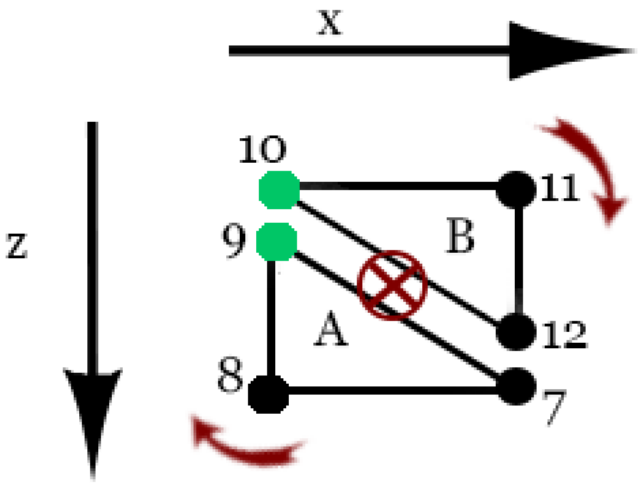
$$Ponto\ 9 = (i + xn,\ j,\ 0)$$

$$Ponto\ 10 = (i,\ j + yn,\ 0)$$

$$Ponto\ 11 = (i + xn,\ j + yn,\ 0)$$

$$Ponto\ 12 = (i + xn,\ j,\ 0)$$

4.4 Face 4



Os índices de iteração nesta face são os mesmos de que na face 2, porém o y toma valor de 0. O ponto de iteração, representado pelos pontos 9 e 10 cujas coordenadas são (i,0,k), auxilia-nos a obter os restantes pontos necessários:

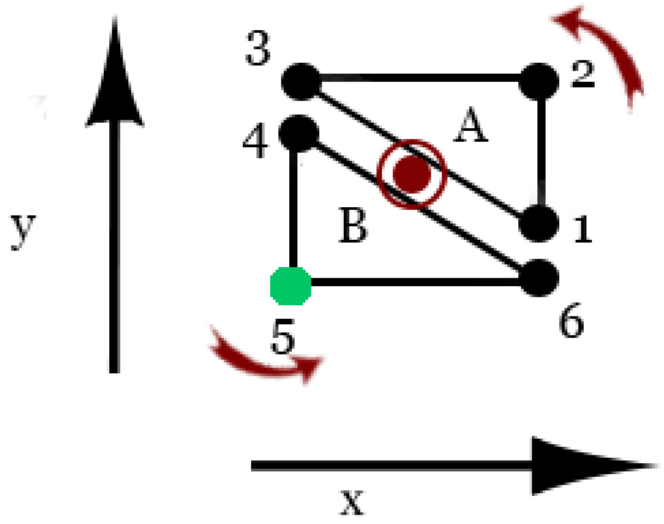
$$Ponto\ 7 = (i + xn,\ 0,\ k + zn)$$

$$Ponto\ 8 = (i,\ 0,\ k + zn)$$

$$Ponto\ 11 = (i + xn,\ 0,\ k)$$

$$Ponto\ 12 = (i + xn,\ 0,\ k + zn)$$

4.5 Face 5



Como dito anteriormente, os índices de iteração serão o j e o i, porém o valor de z será aquele que nos foi dado. Através do ponto atual (ponto 5) cujas coordenadas são (i,j,z) chegamos aos seguintes pontos:

$$Ponto\ 1 = (i + xn,\ j,\ z)$$

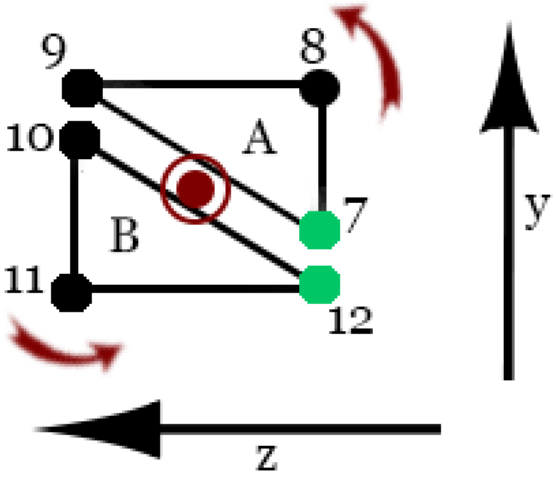
$$Ponto\ 2 = (i + xn,\ j + yn,\ z)$$

$$Ponto\ 3 = (i,\ j + yn,\ z)$$

$$Ponto\ 4 = (i,\ j + yn,\ z)$$

$$Ponto\ 6 = (i + xn,\ j,\ z)$$

4.6 Face 6

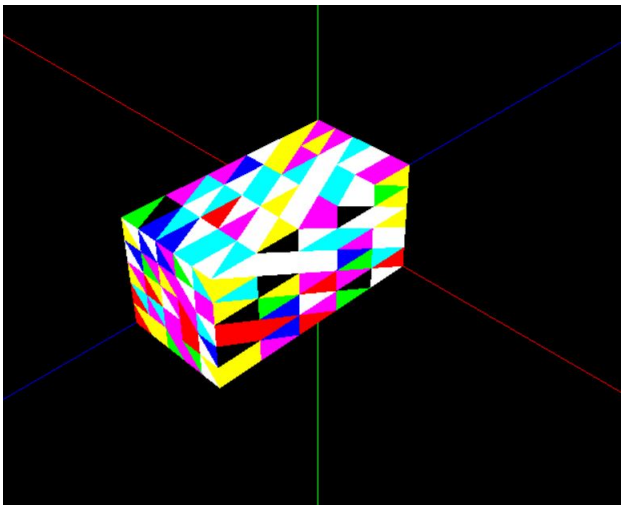


Esta face vai usar os mesmos índices de iteração, uma vez que apenas o valor de z e y mudam e o valor de x não é 0, mas o valor dado. Sendo que o ponto 7 e ponto 12 representam o ponto atual (x,j,k), induzimos os seguintes pontos:

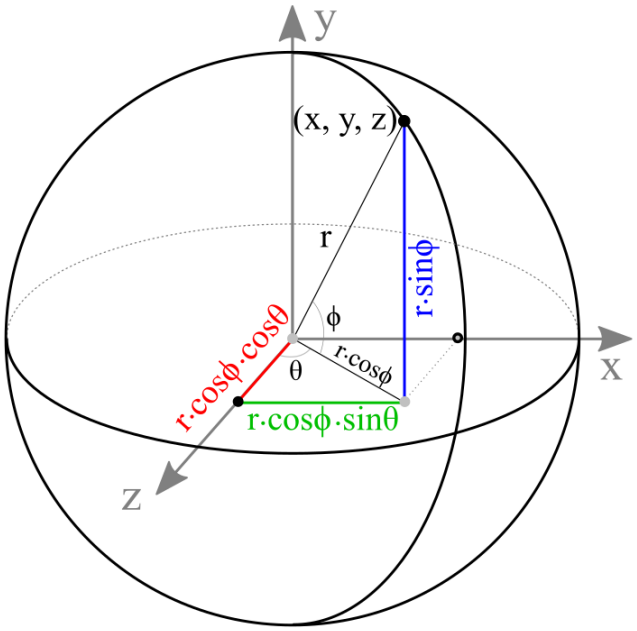
$$Ponto\ 8 = (x,\ j + yn,\ k)$$
$$Ponto\ 9 = (x,\ j + yn,\ k + zn)$$
$$Ponto\ 10 = (x,\ j + yn,\ k + zn)$$
$$Ponto\ 11 = (x,\ j,\ k + zn)$$

Assim, quando faces têm o mesmo comportamento de iteração, o trabalho feito para essas faces vai ser feito nos mesmo *for*s. Também, como é possível observar pelas figuras acima, alguns pontos são iguais, por isso decidimos remover repetidos e reutilizar pontos.

Exemplo de uma *box* com x = 1, y = 1, z = 2 e 4 divisões:



5 Esfera



De modo a gerarmos os pontos necessários para formar uma esfera, vamos iterar a esfera como se fosse uma matriz. Assim, precisamos de saber os índices de iteração. Decidimos portanto, percorrer por número de stacks, em que em cada iteração vamos percorrer também o número de slices.

Para sabermos os pontos em cada iteração, precisamos dos ângulos atuais teta e phi. Para tal, temos ângulos auxiliares stackSkew e sliceSkew resultados das seguintes equações:

$$stackSkew = PI/stacks$$

$$sliceSkew = (2 * PI)/slices$$

Onde stacks e slices representam o número de stacks e slices que nos foram dados, respetivamente.

Assim, estes ângulos vão auxiliar no cálculo dos ângulos atuais phi e teta representados na figura acima. Estes vão mudando em cada iteração e por isso é que são usados os ângulos sliceSkew e stackSkew. Os ângulos dependem então de cada iteração:

$$phi = (PI/2) - (i * stackSkew)$$

$$teta = j * sliceSkew$$

Onde i, j representam os índices de iterações. Como i percorre o número de stacks, o ângulo phi está entre -PI/2 e PI/2 e j percorre o número de slices, o ângulo teta está entre 0 e 2*PI. Assim, com a figura acima e os ângulos atuais teta e phi, é fácil obter o ponto atual:

$$x = raio * cos(phi) * cos(teta)$$

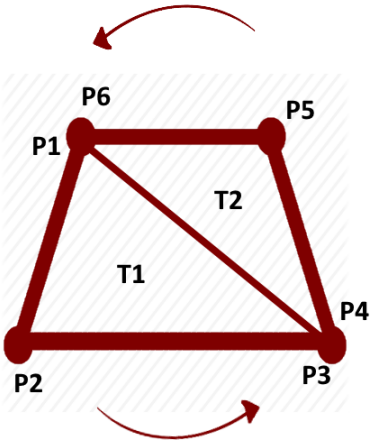
$$y = raio * sin(phi)$$

$$z = raio * cos(teta) * cos(phi)$$

Na imagem abaixo temos um trapézio representado, duma iteração. O ponto atual é representado pelo ponto P2. Em relação ao ponto 2, o ponto 3 difere apenas o valor de z e x. Assim, para obtermos o mesmo temos de fazer uma rotação em torno do eixo y. O ponto 3 pode então ser representado da seguinte forma

$$(P2x*cos(sliceSkew)+P2z*sin(sliceSkew), radio*sin(phi), P2x*-sin(sliceSkew)+P2z*cos(sliceSkew))$$

P_{2x} e P_{2z} *representam as componentes* x *e* z *do ponto* 2.



O cálculo para obtermos o ponto 1 e, por sua vez, o ponto 6, é muito semelhante ao ponto atual (ponto 2), apenas o ângulo que iremos por nos cos e sin vai ser diferente pois queremos o ângulo da iteração anterior. Para tal, temos de somar ao teta o sliceSkew e ao phi o stackSew, obtendo a seguinte equação para o ponto 1:

$$(radius * cos(phi + stackSkew) * sin(teta + sliceSkew), radius * sin(phi + stackSkew),$$

$$radius * cos(teta + sliceSkew) * cos(phi + stackSkew))$$

Tal como foi feita uma rotação a partir do ponto 2 para obter o ponto 3/4, vamos fazer o mesmo para o ponto 5 a partir do ponto 1:

$$(P1x * cos(sliceSkew) + P1z * sin(sliceSkew), radius * sin(phi + stackSkew),$$

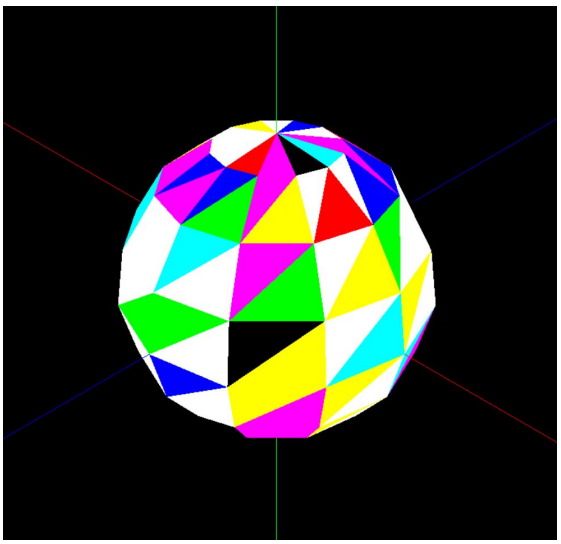
$$P1x * -sin(sliceSkew) + P1z * cos(sliceSkew))$$

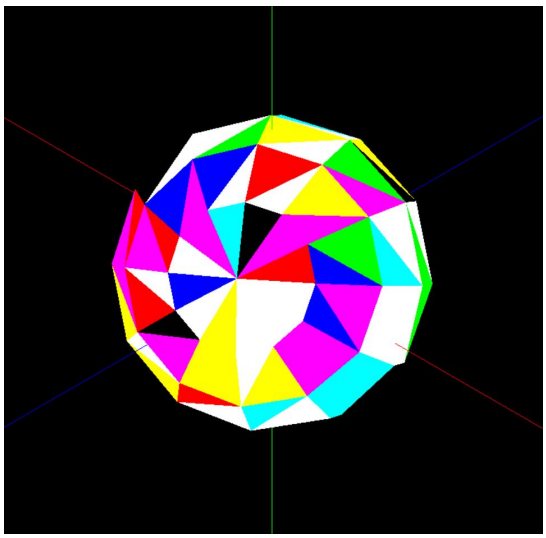
P1x e P1z representam as componentes x e z do ponto 1. Como P1 é igual a P4 e P1 igual a P6, obtemos o seguinte código:

```
void generateSphereFile(double radius, int slices, int stacks, string f) {

    double i, j;
    vector<Triangle> triangles;

    double stackSkew = PI / (stacks), sliceSkew = (2 * PI) / slices;
    for (i = 1; i <= stacks;i++) {
        double phi = (PI / 2) - (i * stackSkew);
        for (j = 1; j <= slices ;j++) {
```

```
            double teta = j * sliceSkew;

            double previousX = radius * cos(phi) * sin(teta);
            double previousZ = radius * cos(teta) * cos(phi);


Point p1(radius * cos(phi + stackSkew) * sin(teta + sliceSkew), radius * sin(phi +
stackSkew), radius * cos(teta + sliceSkew) * cos(phi + stackSkew));
Point p2(previousX, radius * sin(phi),previousZ);
Point p3(previousX * cos(sliceSkew) + previousZ * sin(sliceSkew), radius * sin(phi),
-previousX * sin(sliceSkew) + previousZ * cos(sliceSkew));

Triangle t(p1, p2, p3);
triangles.push_back(t);

double auxX = radius * cos(phi + stackSkew) * sin(teta + sliceSkew),
auxZ = radius * cos(teta + sliceSkew) * cos(phi + stackSkew);

Point p5(auxX * cos(sliceSkew) + auxZ * sin(sliceSkew), radius * sin(phi + stackSkew),
-auxX * sin(sliceSkew) + auxZ * cos(sliceSkew));

//p4=p3 e p6=p1
Triangle t1(p3, p5, p1);
triangles.push_back(t1);

        }
    }
    trianglesToFile(triangles, f);
}
```

Exemplo de uma esfera com raio (*radius*) 2, 12 *slices* e 8 *stacks*:





6 Cone

Analisando os argumentos da função que cria o *file* com a informação para gerar o cone, é dado o raio da base do cone, a altura do cone, o número de *slices* e o número de *stacks* que o constituem.

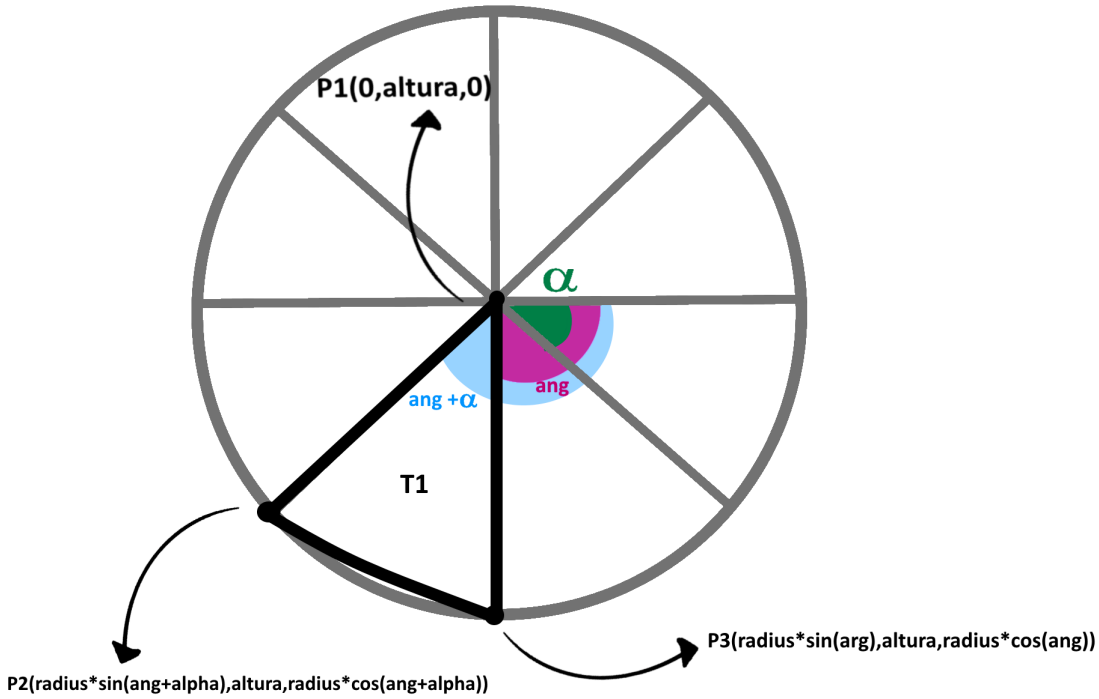
Como primeira consideração no que toca às suas variáveis, de assinalar que ***altura*** designa o ponto do eixo dos *yy* no qual se encontra a base do cone, sendo o valor negativo de metade da altura (***height***) dada como argumento, dado que o sólido estará centrado na origem do referencial.

```
void generateConeFile(double radius, double height, double slices, double stacks, string f)

    vector<Triangle> triangles;

    double alpha = (2 * PI) / slices;
    double beta = height / stacks;
    double altura = -height / 2;
    double i, j;
    double ang, raio1, raio2, frst, scnd;
```

O código seguinte gera o círculo que será a base do cone.

```
 for (i = 0; i < slices; i++) {

        ang = alpha * i;

        Point p1(0, altura, 0);
        Point p2(radius * sin(ang + alpha), altura, radius * cos(ang + alpha));
        Point p3(radius * sin(ang), altura, radius * cos(ang));

        Triangle t1(p1, p2, p3);
        triangles.push_back(t1);
    }
```

Se for dado como argumento 8 no número de slices, que estamos na iteração do ciclo *for* em que i=2 e considerando, confrontando o código acima, uma variável **ang** tal que que **ang = alpha*i**, este foi o esquema que criamos e de que partimos para obter as coordenadas dos pontos dos triângulos que constituem a base do nosso cone.

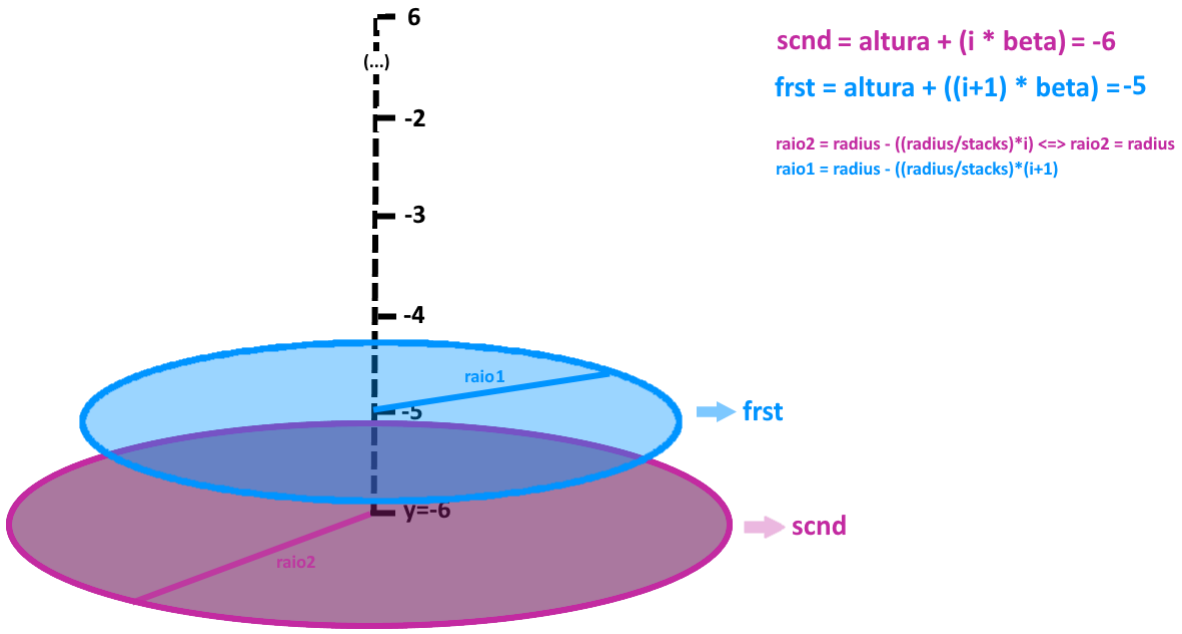


É a seguir apresentada a nossa abordagem no que toca à criação da restante parte do cone.

```
 for (i = 0; i < stacks; i++) {
        scnd = altura + (i * beta);
        frst = altura + ((i + 1) * beta);

        raio2 = radius - ((radius / stacks) * i);
        raio1 = radius - ((radius / stacks) * (i + 1));

        for (j = 0; j < slices; j++) {
            height = alpha * j;

            Point p4(raio2 * sin(height), scnd, raio2 * cos(height));
            Point p5(raio1 * sin(height + alpha), frst, raio1 * cos(height + alpha));
            Point p6(raio1 * sin(height), frst, raio1 * cos(height));

            Triangle t2(p4, p5, p6);
            triangles.push_back(t2);

            Point p7(raio2 * sin(height + alpha), scnd, raio2 * cos(height + alpha));

            Triangle t3(p4, p7, p5);
            triangles.push_back(t3);
        }
    }
    trianglesToFile(triangles, f);
}
```

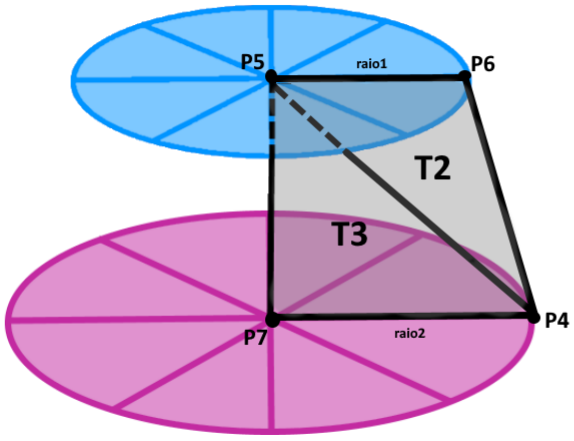
Supondo a criação de um cone com *height* (altura do cone) 12, 6 stacks e 8 slices (mesmo número de *slices* que na explicação da base do cone), procura-se de seguida explicar visualmente a origem das fórmulas que usamos para a criação.

Relembrando a variável e **beta = height / stacks**, imaginemos a situação em que o ciclo exterior se encontra na sua primeira iteração, com i=0.



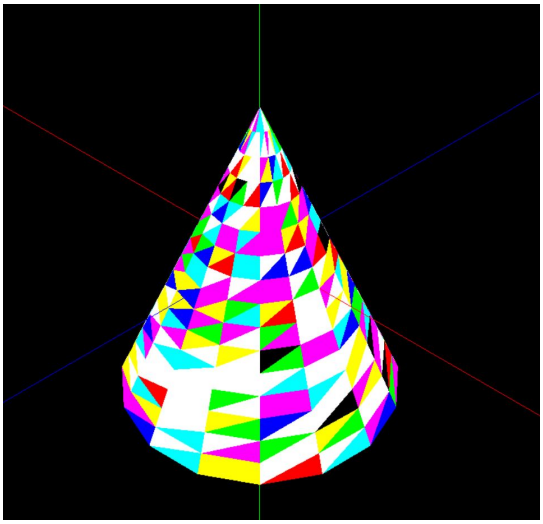
A mesma lógica aplicar-se-á para as restantes *stacks*, com os raios a diminuírem sucessivamente.

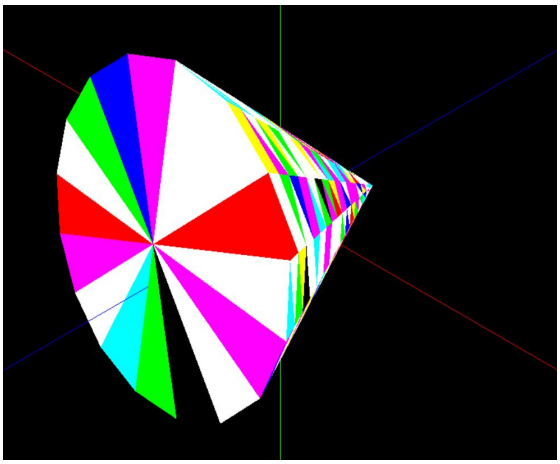
Para cada uma das stacks (ciclo exterior), com o ciclo interior originamos os triângulos que constituem a porção do cone entre as duas camadas (*frst* e *scnd*), **para cada uma das *slides* que constituem o cone**. O raciocínio usado para deduzirmos os pontos que formam os triângulos é representado no esquema seguinte:



P4 = raio2*sin(height) , scnd, raio2*cos(height)
P5 = raio1*sin(height+alpha), frst,raio1*cos(height+alpha)
P6 = raio1*sin(height), frst, raio1*cos(height)
P7 = raio2*sin(height+alpha), frst, raio1*cos(height+alpha)

Exemplo de um cone com raio(*radius)* 2, altura(*height*) 4, 16 *slices* e 16 *stacks*:





7 XML

O *Engine* (representado pela classe 2) é a aplicação responsável de ler ficheiros *xml*, guardar em memória os pontos e fazer *display* dos modelos, com possibilidade de algumas *features* referidas no *README*. Implementamos uma classe *Models* que contém uma lista de nomes de ficheiros, um *int* que representa a posição de leitura do ficheiro, um *int* representando o comprimento deste *array* e uma *hashtable*. Esta tem como *key* o nome do ficheiro onde o modelo é guardado e o *value* é uma lista de listas. Isto porque guardamos um triângulo em cada linha. Assim, cada linha tem uma lista com o respetivo triângulo (já definido com 9 pontos e cor para ser mais acessível com *OpenGL*) dado a lista de listas.

Assim, para fazermos *display* de todos os modelos, vamos percorrer o *array*, buscar à *hashtable* os pontos do respetivo ficheiro e aumentámos a posição. Fazemos *display* do ficheiro no índice resto da divisão entre posição e comprimento, de modo a voltar ao modelo inicial.

Definimos o nome *"Files.xml"* para o ficheiro que conterá os modelos. Para escrita e leitura deste ficheiro, recorremos a *tinyxml2*. Quando um modelo é criado através do generator é chamada uma função updateXML que verifica se o ficheiro já existe. Se sim, verifica ainda se já existe um ficheiro com o mesmo nome que nos foi dado, impossibilitando escrever por cima se apresentam o mesmo nome. Se não tiver ficheiro com o mesmo nome, adiciona à cena. Se não existir o ficheiro, será criado um ficheiro xml com o mesmo nome acima e adicionado o modelo ao mesmo.

A leitura do ficheiro xml é realizada apenas uma vez quando corrido o engine. Se existir o ficheiro "Files.xml" então o programa irá percorrer todos os filhos retirar o nome do atributo (file) e chamar o método addFile da classe Models que trata de adicionar ao array com o nome de ficheiro e à hashtable.

```
<Scene>
    <Model file="sphere.3d"/>
    <Model file="cone.3d"/>
</Scene>
```

8 Conclusões

Esta primeira fase do trabalho permitiu-nos uma maior familiarização com a ferramenta OpenGl e a implementação das suas funções dos sólidos pré-definidas. No que toca ao cumprimento das tarefas propostas, consideramos que foi comprido o objetivo definido.