Universidade do Minho
Escola de Engenharia

Computação Gráfica

TRABALHO PRÁTICO

1ª Fase

Mestrado Integrado em Engenharia Informática

Fábio Quintas Gonçalves, a78793

Francisco José Moreira Oliveira, a78416

Raul Vilas Boas, a79617

Vitor Emanuel Carvalho Peixoto, a79175

Ano letivo 2017/2018

Março de 2018

ÍNDICE

1. Introdução 1
2. *Generator* 2
   2.1 Plano 2
   2.2 Caixa 3
   2.3 Esfera 4
   2.4 Cone 7
   2.5 Extras 8
3. *Engine* 10
   3.1 Leitura dos ficheiros XML 10
   3.2 Desenho e apresentação do modelo 10
   3.3 *Debug* e câmara 10
4. Conclusões e trabalho futuro 12

1. INTRODUÇÃO

Este relatório explicita a construção de duas aplicações relativas à 1ª fase deste trabalho prático: o *generator* e o *engine*.

O *generator* gera ficheiros com a informação dos modelos que que pretendemos produzir, sendo que neste caso gera apenas os pontos dos vértices dos modelos. O *engine* irá ler um ficheiro de configuração (escrito em *XML*), produzindo e apresentando os modelos anteriormente gerados.

Nesta fase do projeto, as seguintes primitivas gráficas foram desenvolvidas:

- Plano
- Caixa
- Esfera
- Cone
- Pirâmide
- Cilindro

No resto deste relatório, iremos explicitar como o *generator* foi desenvolvido e como cada modelo foi desenvolvido, assim como uma descrição do *engine* e do modo de apresentação e visualização dos modelos, de forma a realizar o pretendido das duas aplicações.

2. *GENERATOR*

O *generator,* começa por receber, como primeiro parâmetro, o tipo da primitiva que queremos gerar. Conforme esta primitiva, o número dos outros parâmetros a receber varia, sendo estes necessários para a criação do modelo pretendido. Por fim, o último parâmetro é o nome do ficheiro (<exemplo>.3d) que o *generator* vai criar com os pontos gerados.

Estes pontos gerados representam os vértices das figuras, o que será o necessário para, futuramente, o *engine* desenhar a figura pretendida.

2.1 Plano

Para criar um plano em *OpenGL* é necessário a junção de dois triângulos. Para isso é necessário fornecer a largura do plano, para assim pudermos saber as coordenadas dos pontos que levarão à criação do respetivo triângulo. Logo, efetuou-se o seguinte algoritmo para descobrir os pontos respetivos do plano do tamanho dado.

Algoritmo:

Função *plane* com a largura do plano.

```
Criar a variável lado que é igual à largura a dividir por dois.
Criar um vetor pontos onde se vai guardar os pontos gerados.
//Criar o primeiro triângulo
    P(lado, 0.0, lado)
    P(lado, 0.0, -lado)
    P(-lado, 0.0, lado)
//Criar o segundo triangulo
    P(-lado, 0.0, -lado)
    P(-lado, 0.0, lado)
    P(lado, 0.0, -lado)
Imprimir os pontos criados num ficheiro.
```

Fim do algoritmo.

Os cuidados que se teve na criação do algoritmo foi na ordem dos pontos, pois eles têm de seguir a regra da mão direita, para assim o plano poder ser visível, isto é, estar "virado" para cima.

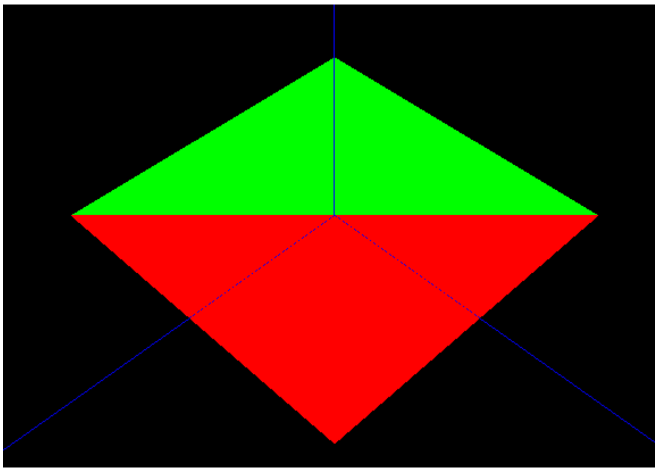


Figura 1 – Inserir legenda.

2.2 Caixa

Para representar a caixa são necessários 6 planos, logo é preciso 12 triângulos, no entanto como a função também vai receber o número de divisórias vai implicar a criação de planos mais pequenos dependendo do número de divisões.

Algoritmo:

Função *box* com as coordenadas do x, y, z e o numero de divisões.

```
Criação do vetor pontos onde se vai guardar os pontos gerados.
Criar os pontos da face frontal
    Criar a variável px e py que têm o valor de -x e y, respetivamente.
    Enquanto que px é menor que x faz:
        //Criar o primeiro triângulo
            P(px, py, z/2)
            P(px, py – (y/nDivisoes), z/2)
            P(px + (x/nDivisoes), py, z/2)
        //Criar o segundo triângulo
            P(px, py – (y/nDivisoes), z/2)
            P(px + (x/nDivisoes), py – (y/nDivisoes), z/2)
            P(px + (x/nDivisoes), py, z/2)
    Atualizar o valor do py -= y/nDivisoes
Atualizar o valor do px += x/nDivisoes
```

Fim do algoritmo.

A figura apresentada em baixo representa a construção da face frontal apresentada no pseudocódigo, para assim consolidar melhor a ordem na qual foram efetuadas a criação dos planos.

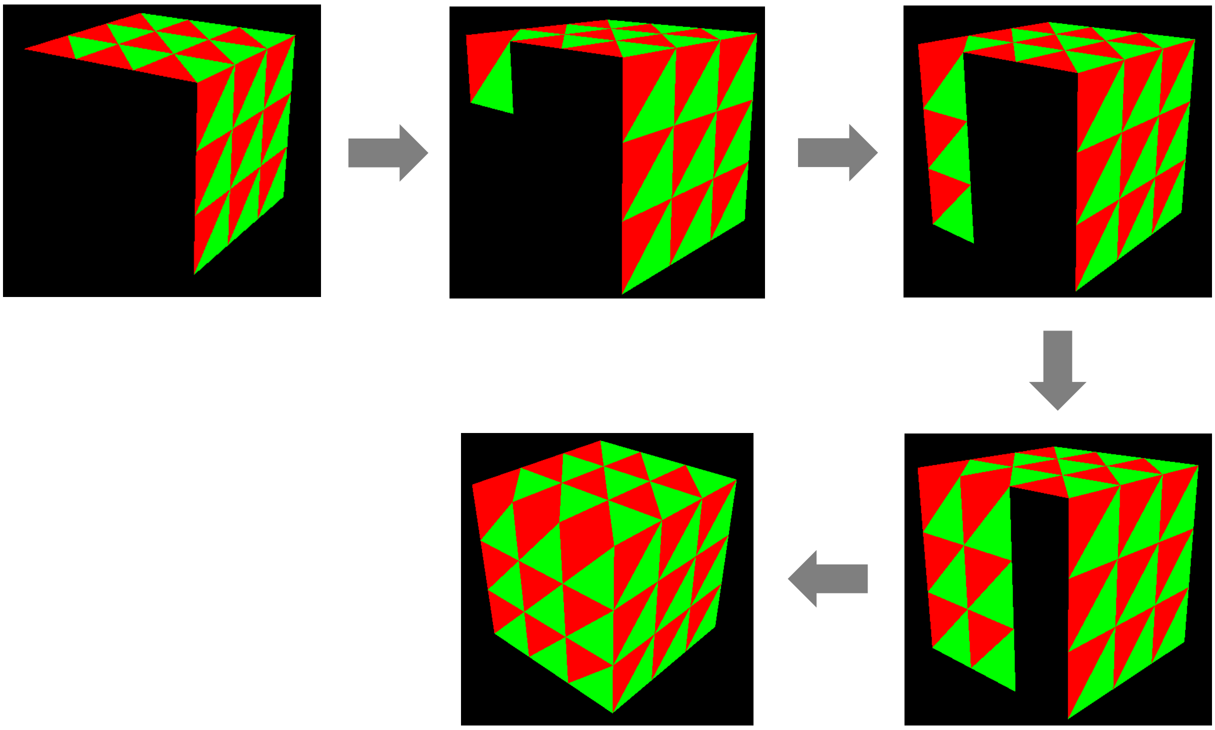


Figura 2 – Evolução de uma face de um cubo.

Para as outras faces usou-se a mesma estratégia que o algoritmo anterior, apenas variando as variáveis usadas dependendo do plano a ser criado.

2.3 Esfera

Para desenharmos a esfera, é necessário recorrer a um método de cálculo de coordenadas de pontos à sua superfície. Neste caso em específico, essas coordenadas irão ser determinadas usando a informação existente: raio do equador da esfera e os ângulos α e β, sendo que α é o ângulo na superfície circular do equador e β é o ângulo desde essa superfície circular até à superfície da esfera, como retrata a seguinte imagem:

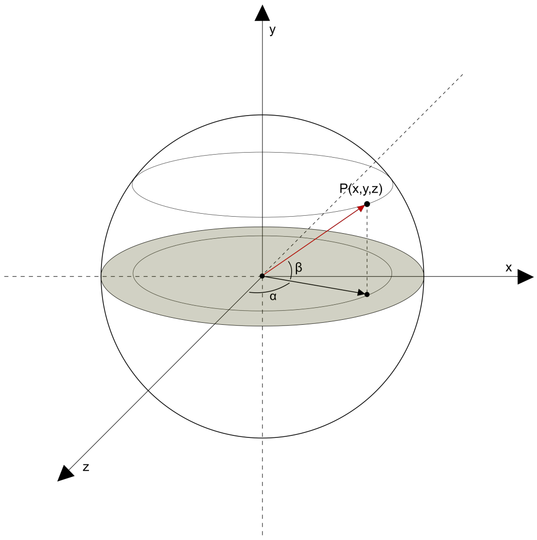


Figura 3 – Coordenadas de um ponto na superfície de uma esfera.

Com efeito, as coordenadas do ponto P podem ser calculadas da seguinte forma:

$$x = r * \cos(\beta) * \sin(\alpha)$$

$$y = r * \sin(\beta)$$

$$z = r * \cos(\beta) * \cos(\alpha)$$

Dado que os argumentos são o raio, *slices* e *stacks* que a esfera deverá ter, é necessário trocar os valores de α e β, relacionando estes com as *slices* e *stacks*.

De facto, o valor de α pode ser obtido através das *slices*. Seguindo a figura 1, verificamos que α circula sobre um círculo de 360º de amplitude (2π em radianos), logo α irá ser incrementado a cada *slice* em $2\pi/slices$.

β, por outro lado, irá depender das *stacks*, podendo variar longitudinalmente na esfera (como se de um meridiano se tratasse, logo através de 90º), assim $\beta = \pi/stacks$.

Um dos problemas de fazer uma esfera recorrendo a *slices* e *stacks* deve-se aos polos da esfera.

Se repararmos, a primeira e ultima *stack* de uma esfera serão diferentes das restantes, visto que os triângulos que compõem essa *stack* terão um vértice comum, o topo ou o fundo, respetivamente.

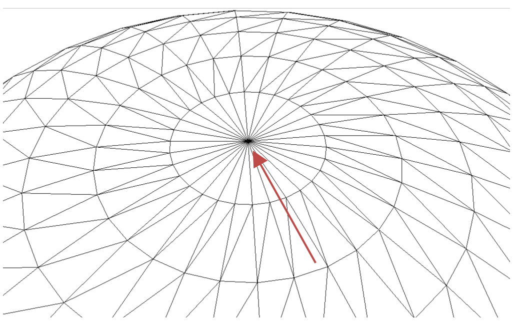


Figura 4 – Vértice comum de todos os triângulos da *stack* do topo.

Algoritmo:

Função *sphere* com o raio, *stacks* e *slices*.

```
Criação do vetor pontos onde se vai guardar os pontos gerados.
Criar os pontos da face frontal
Variável alfa é igual a 2*π/slices.
Variável beta é igual a π/stacks.
Enquanto que a variável slice, inicializada a zero for menor que slices faz:
    //Topo da esfera:
        Calcular os 3 pontos a partir das fórmulas
    //Base da esfera
        Calcular os 3 pontos a partir das formulas
    Enquanto que stack, inicializada a zero, for menor que stacks faz:
        //Metade superior
            //Triângulo inferior
                Calcular os 3 pontos a partir das fórmulas
            //Triângulo superior
                Calcular os 3 pontos a partir das fórmulas
        //Metade inferior
            //Triângulo inferior
                Calcular os 3 pontos a partir das fórmulas
            //Triângulo superior
                Calcular os 3 pontos a partir das fórmulas
        Valor de stack incrementa.
    Valor de slice incrementa.
```

Fim do algoritmo.

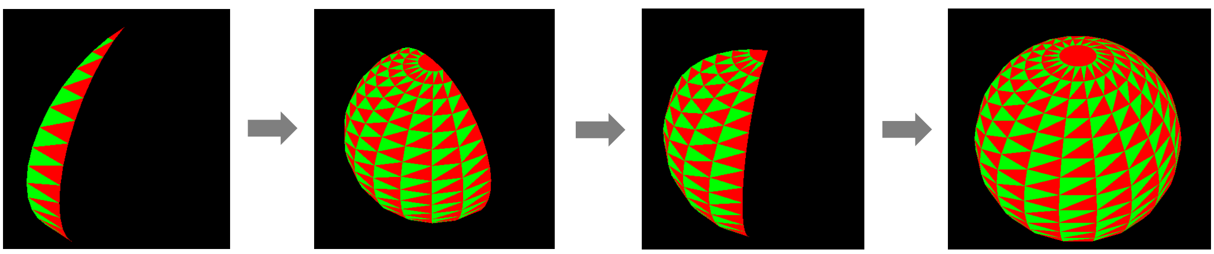


Figura 5 – Evolução de uma esfera.

2.4 Cone

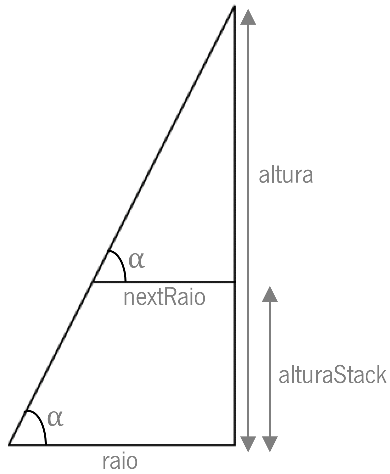
Para a construção do cone foi dividida a figura em três partes. A base que é um circulo, os lados do cone que segue uma estrutura parecida à do circulo excetuando a formula como as coordenadas dos pontos são obtidas e por último o topo do cone.

Para obter as coordenadas dos pontos nas diferentes *stacks* tivemos de obter a expressão que iria sempre obter o raio da *stack* em questão. Pois, apenas com esse valor é que conseguimos descobrir os pontos da circunferência (px = raio * sin(*alfa*) e pz = raio * cons(*alfa*)). Para isto, também era necessário saber a altura da *stack* considerada, por esse motivo, dividiu-se a altura do cone pelas *stacks*, para assim sabermos a distância a que cada *stack* está uma da outra.

Sendo assim realizou-se os seguintes cálculos para obter a expressão.

$$\tan(\alpha) = \sin(\alpha) / \cos(\alpha)$$
$$\Leftrightarrow \tan(\alpha) = \text{altura}/\text{raio}$$
$$\Leftrightarrow \alpha = \tan^{-1}(\text{altura}/\text{raio})$$

$$\tan(\alpha) = (\text{altura} - \text{alturastack})/\text{nextRaio}$$
$$\Leftrightarrow \tan(\tan^{-1}(\text{altura}/\text{raio})) = (\text{altura} - \text{alturastack})/\text{nextRaio}$$
$$\Leftrightarrow (\text{altura}/\text{raio}) = (\text{altura} - \text{alturastack})/\text{nextRaio}$$
$$\Leftrightarrow \text{nextRaio} = (\text{altura} - \text{alturastack}) * \text{raio}/\text{altura}$$



Com estes cálculos obtemos assim a fórmula que nos fornece o valor do raio em qualquer stack.

Algoritmo:

Função *cone* com o raio, altura, *stacks* e *slices*.
Criação do vetor pontos onde se vai guardar os pontos gerados.
Variável alfa é igual a 2*π/slices.
Variável beta é igual a π/stacks.
Enquanto que slice, variável inicializada a 0, for menor que slices faz:
//Circulo da base:
Calcular os 3 pontos com as formulas das coordenadas polares.
Enquanto stack, variável inicializada a 0, for menor que stacks, faz:
Criar varável alturaStack que representa a stack atual.
Criar variável novoRaio que representa o raio atual.

```
        Criar variável nextStack que representa a altura da próxima stack.
        Criar variável nextRaio que representa o tamanho do próximo raio.
        //Triângulo de cima
            Calcular os 3 pontos com as fórmulas das coordenadas polares.
        //Triângulo de baixo
            Calcular os 3 pontos com as fórmulas das coordenadas polares.
        Incrementar valor da stack.
        //Ponta do cone
            Calcular os 3 pontos da ponta do cone.
    Incrementar o valor da slice.
```

Fim do algoritmo.

Na figura em baixo mostra-se então a evolução da construção do cone ao longo dos ciclos.

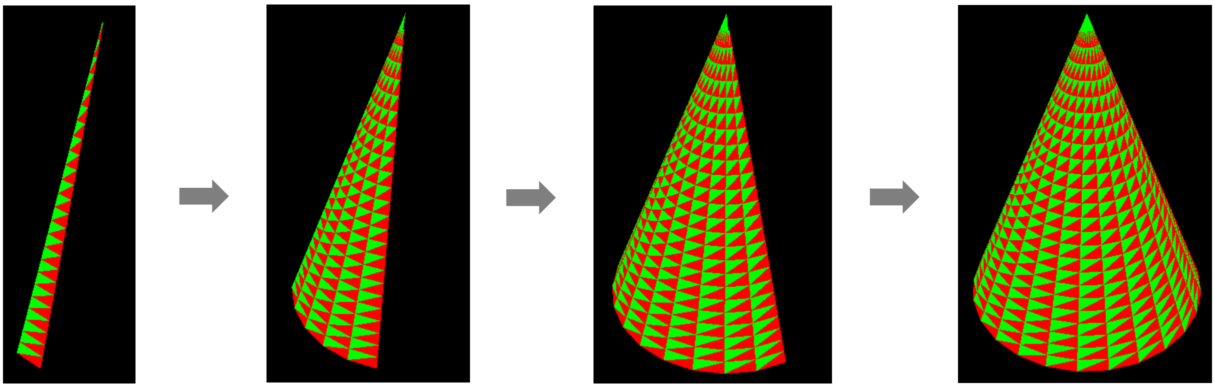


Figura 6 – Evolução de um cone.

2.5 Extras

Como extras criou-se um circulo, uma pirâmide com divisões e um cilindro. Estas figuras foram criadas mais com o intuito de ajudar a entender as outras figuras pois, por exemplo, o circulo foi usado como base do cone. Sendo que, os seus algoritmos não mudavam muito das figuras explicadas anteriormente.

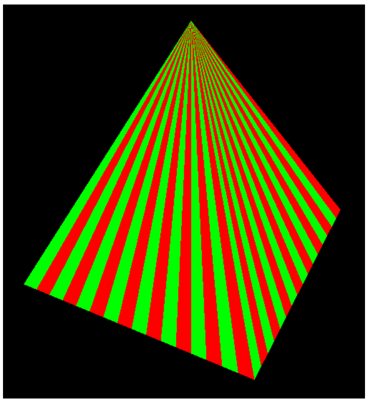


Figura 7 – Pirâmide.

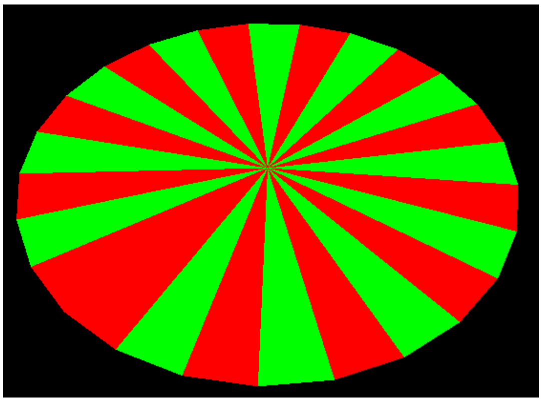


Figura 8 – Circulo.

Figura 9 – Cilindro.

3. *ENGINE*

3.1 Leitura dos ficheiros XML

A primeira fase da *engine* passa por carregar os ficheiros necessários para a renderização. Para tal primeiramente carrega-se o ficheiro *scene.xml*, onde utilizamos como *parser* a biblioteca *tinyxml2*, que vai comunicar à *engine* quais os ficheiros *.3d*, que contém os pontos previamente gerados pelo *generator,* necessários a carregar para geração das figuras.

3.2 Desenho e apresentação do modelo

Tendo já os devidos ficheiros *.3d* carregados passa-se à fase de apresentação das figuras. Os ficheiros *.3d* têm em cada linha as coordenadas espaciais de um ponto e cada conjunto de 3 pontos representa um triangulo. Sabendo isto, fazemos o devido processamento das linhas do ficheiro e são gerados triângulos com a utilização do *glBegin(GL_TRIANGLES)* que a cada 3 instancias de *glVertex*, onde colocamos os pontos presentes em cada linha e desenha graficamente o respetivo triângulo. O resultado final são as figuras pretendidas.

3.3 *Debug* e câmara

De modo a ajudar no nosso desenvolvimento das figuras, assim como para uma melhor apresentação das figuras, implementamos uma câmara de modo a poder mover à volta da figura, assim como diferentes modos de visualização.

Adicionamos um eixo interativo (pressionando a tecla 'z' no teclado) para podermos visualizar a figura com os eixos X, Y e Z para uma melhor compreensão do estado e posicionamento da figura.

Para a câmara, usamos coordenadas esféricas para posiciona-la em torno da figura. Essencialmente, estamos a criar uma esfera em torno do centro, sendo que a câmara olha para dentro da esfera.

Se quisermos nos aproximar da figura, ou seja, do centro, podemos diminuir o raio da esfera, tornando-a mais pequena (e de igual modo, podemos aumentar o raio para nos afastarmos da esfera). Aumentando ou diminuindo α e β, podemos alterar a posição onde a câmara se posiciona na superfície da esfera.

É de notar que a implementação das coordenadas esféricas, assim como a sua transformação para coordenadas cartesianas foram efetuadas de igual modo ao já especificado na secção 2.3 deste relatório. Com a função *glPolygonMode(face, option)*, alterando o *option* para opções como *GL_POINT, GL_LINE, e GL_FILL*, podemos apresentar o modelo com apenas os seus pontos, apenas as suas linhas e a figura completa, respetivamente.

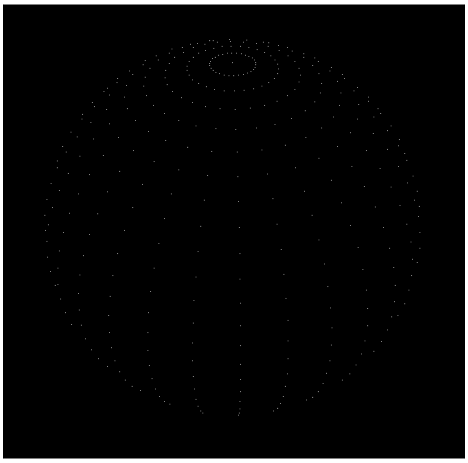


Figura 10 – *GL_POINT* numa esfera.

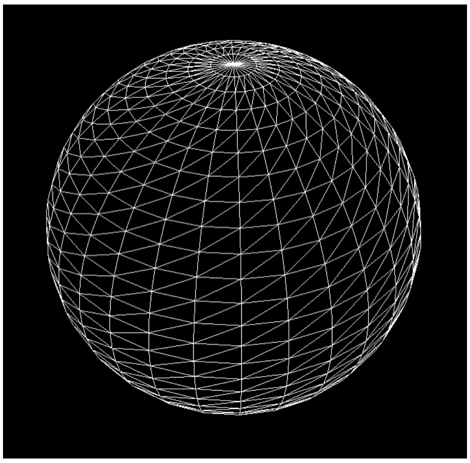


Figura 11 – *GL_LINE* numa esfera.

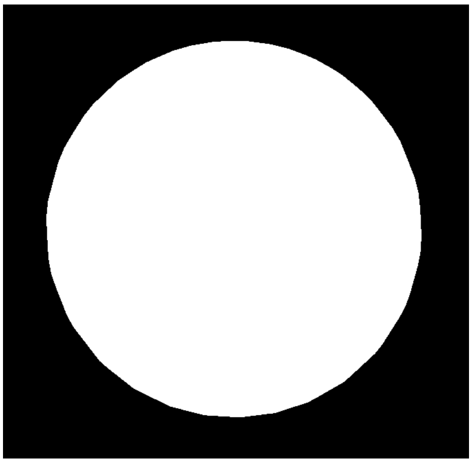


Figura 12 – *GL_FILL* numa esfera.

4. Conclusões e trabalho futuro

Esta primeira fase permitiu absorver conhecimento e técnicas importantes no que toca à utilização de do *OpenGL* e a sua integração com a linguagem de programação C++. Para além disso permitiu-nos traçar um caminho inicial para o desenvolvimento das restantes fases deste trabalho prático, através do *engine* e do *generator*.

Futuramente esperamos que o trabalho desenvolvido até agora permita acelerar o projeto a ser desenvolvido, quer devido ao facto de já termos ferramentas prontas, quer ao conhecimento que trazemos, fruto da prática desenvolvida nesta primeira fase.