Universidade do Minho
Mestrado Integrado em Engenharia Informática
Departamento de Informática

Computação Gráfica

Graphical Primitives

Isabel Sofia da Costa Pereira A76550
José Francisco Gonçalves Petejo e Igreja Matos A77688
Maria de La Salete Dias Teixeira A75281
Tiago Daniel Amorim Alves A78218

12 de Março de 2018

Conteúdo

1 Introdução

No âmbito da UC de Computação Gráfica foi-nos proposto a realização de duas aplicações. Sendo a primeira um gerador e a segunda um motor. Para o desenvolvimento destas aplicações foi necessário utilizar certos recursos tais como C++ e OpenGL.

O gerador necessita de receber o nome de uma figura primitiva, os parâmetros desta e o nome de um ficheiro. Desta forma, esta aplicação tem como objetivo gerar todos os vértices necessários para a elaboração dos triângulos que constituem a figura indicada no argumento e guardá-los no ficheiro estabelecido.

O motor é responsável por analisar o conteúdo de um ficheiro XML verificando quais os nomes de ficheiros que este contém. De seguida, o motor abre os ficheiros e desenha os vértices contidos nestes, obtendo-se assim uma figura permitiva.

As figuras primitivas abordadas neste projeto foram planos, caixas, cones e esferas.

2 Desenvolvimento do Projeto

Para o desenvolvimento do trabalho foi conveniente criar duas classes principais, o *generator* e o *engine*, que representam as duas aplicações requeridas. Como auxilio destas foi necessário a utilização de outras classes como *Point*, *vertex*, *tinyxml2* e *Parser*.

2.1 Gerador

O gerador, *generator*, tal como mencionado anteriormente, é responsável pelo cálculo dos vértices. Desta forma, foi necessário desenvolver algoritmos que, dado as medidas fundamentais para a criação de uma figura, fossem capazes de gerar os vértices necessários para o desenho dos triângulos que constituem a figura em questão.

Como o gerador tem que trabalhar com vértices consideramos conveniente implementar a classe *Point* que representa as coordenadas x, y e z de um vértice. Além dessa classe e para uma melhor organização do codigo, definiu-se também a classe *vertex* onde se pode encontrar o raciocínio para a geração automática dos vértices para um plano, uma caixa, um cone e uma esfera.

Após o cálculo de todos os vértices das figuras requeridas, o gerador escreve num ficheiro em específico esses vértices. Cada ficheiro, independetemente da figura pedida, rege-se pela mesma estrutura. Cada linha possui um vértice, isto é, as coordenadas x, y e z. Por sua vez, estas encontram-se separadas por espaços. Para além disso, como um ficheiro pode conter várias figuras do mesmo género, para facilitar a identificação da passagem de uma figura para a seguinte, os vértices de cada uma destas são limitados por uma linha com o formato - - - *NEW* - - -.

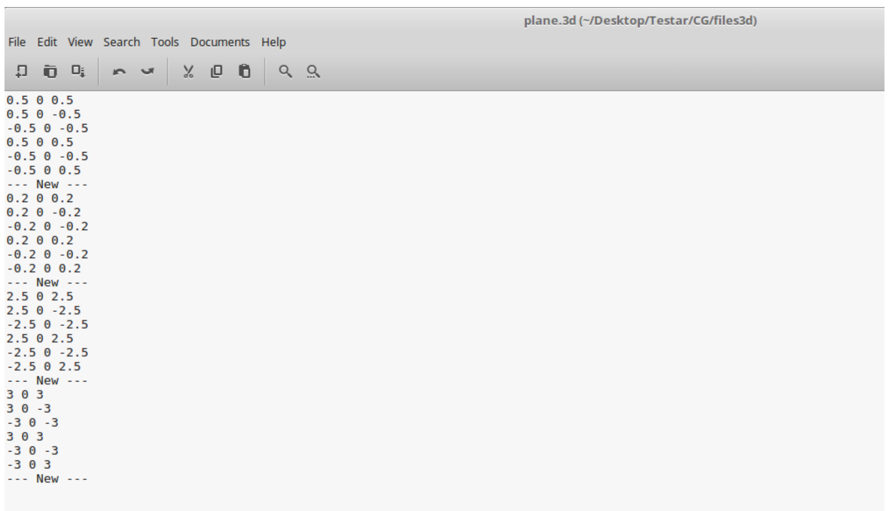

Figura 1: Exemplo da estrutura dos ficheiros 3d

2.1.1 Plano

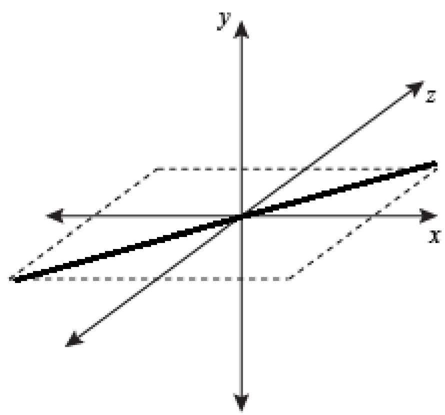

Figura 2: Plano - Desenho

Como se pode constatar na imagem acima, a ideia base do plano passa pelo desenho de dois triângulos com dois vertíces coincidentes. Visto que se trata de um plano XZ, mantém-se a cordenada Y de todos os vertíces a zero.

O valor *size* do plano é passado como argumento indicando o tamanho do plano desejado. Assim, para centrar o plano na origem, esse valor é dividido por dois, obtendo-se dessa maneira as posições positivas ou negativas das coordenadas X e Z de forma a uniformar ambos os triângulos, formando o plano desejado.

2.1.2 Caixa

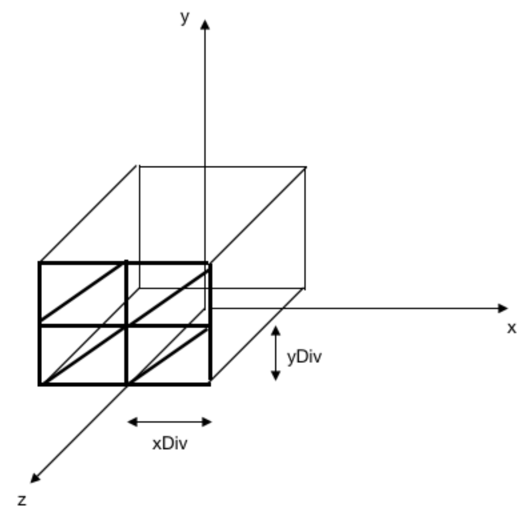

Figura 3: Caixa

A figura acima representa um exemplo de uma caixa com duas divisões, neste caso só numa das faces. Para construir essa face, é necessário ter em conta as dimensões da caixa e o número de divisões que são passados como parâmetros.

Para o cálculo de todos os vértices pertencentes à caixa, foi necessário o desenvolvimento de um algoritmo que interpreta cada uma das faces como se fossem matrizes com n divisões. Para isso, calculámos as coordenadas xDiv = x/div, e yDiv = y/div, sendo x a dimensão da caixa no eixo do X, y a dimensão da caixa no eixo do Y, e div o número de divisões passadas como argumento. Estes dois valores, xDiv e yDiv, são fundamentais para o processo de construção das faces porque representam os desvios que devem ser efetuados na passagem de um vértice para o outro. Para percorrer matrizes implementámos dois ciclos (linha e coluna), e multiplicando os índices de cada um dos ciclos por xDiv e yDiv corretamente, obtém-se a face com as divisões desejadas. Por exemplo, para este caso com 2 divisões e sendo x=4, y=4, z=4, tem-se xDiv=2 e yDiv=2. Começando em (-2,0,2), para obter o vértice (0,0,2), efetua-se a operação (-2+(xDiv*1), 0,2). Pelo mesmo raciocínio, para obter o último vértice (2,0,2) calcula-se (-2+(xDiv*2),0,2).

2.1.3 Esfera

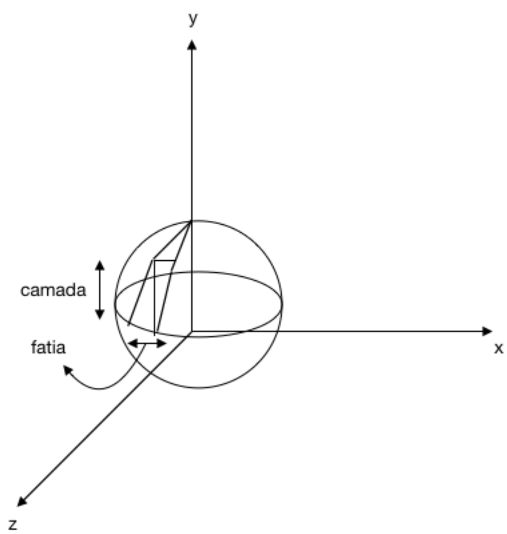

Figura 4: Esquema da Esfera

Com o objetivo de desenhar a esfera com um determinado raio, e dividida por um determinado número de fatias e camadas, foi necessária a utilização das fórmulas que transformam coordenadas polares em coordenadas cartesianas, sendo estas as seguintes:

$$x = radius * cos(h2) * sin(h)$$

$$y = radius * cos(h)$$

$$z = radius * sin(h2) * sin(h)$$

onde,

$$h = \pi/stacks$$

$$h2 = 2\pi/slices$$

O algoritmo divide a esfera em três partes diferentes: topo, base e meio. Com dois ciclos, os triângulos que constituem o topo da esfera são construídos apenas na primeira iteração do ciclo interior, que corresponde à primeira camada; os triângulos que constituem a base da esfera apenas na última iteração do ciclo interior, correspondente à última camada. Em relação ao meio da esfera, é visível no excerto de código abaixo o algoritmo implementado.

```
v.push_back(new Point(
radius*cos((i+1)*h2)*sin((j+1)*h),
radius*cos((j+1)*h),
radius*sin((i+1)*h2)*sin((j+1)*h)));

v.push_back(new Point(
radius*cos((i+1)*h2)*sin((j+2)*h),
radius*cos((j+2)*h),
radius*sin((i+1)*h2)*sin((j+2)*h)));

v.push_back(new Point(
radius*cos(i*h2)*sin((j+1)*h),
radius*cos((j+1)*h),
radius*sin(i*h2)*sin((j+1)*h)));

v.push_back(new Point(
radius*cos(i*h2)*sin((j+1)*h),
radius*cos((j+1)*h),
radius*sin(i*h2)*sin((j+1)*h)));

v.push_back(new Point(
radius*cos((i+1)*h2)*sin((j+2)*h),
radius*cos((j+2)*h),
radius*sin((i+1)*h2)*sin((j+2)*h)));

v.push_back(new Point(
radius*cos(i*h2)*sin((j+2)*h),
radius*cos((j+2)*h),
radius*sin(i*h2)*sin((j+2)*h)));
```

As linhas que estão a efetuar operações como (i+1)*h2, servem para calcular as coordenadas do próximo vértice, efetuando o deslocamento pelo ângulo h2. O mesmo se aplica às deslocações através do ângulo h.

2.1.4 Cone

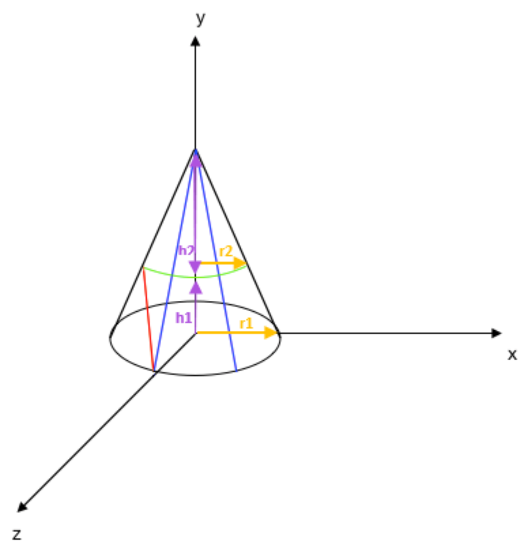

Figura 5: Esquema do Cone

Com o objetivo de desenhar um cone dividido em fatias e em camadas, recorreu-se às seguintes equações, que permitem calcular as alturas, raios e ângulos necessários, bem como transformar coordenadas polares em cartesianas.

$$h = height/stacks$$

$$tan(\beta) = height/radius$$

$$h2 = h * nrStack$$

$$radius2 = (height - h2)/tan(\beta)$$

$$\alpha = 2\pi/slices$$

$$x1 = radius * sin(\alpha * nrSlice)$$

$$z1 = radius * cos(\alpha * nrSlice)$$

$$x2 = radius * sin(\alpha * (nrSlice + 1))$$

$$z2 = radius * cos(\alpha * (nrSlice + 1))$$

$$x3 = radius2 * sin(\alpha * nrSlice)$$
$$z3 = radius2 * cos(\alpha * nrSlice)$$
$$x4 = radius2 * sin(\alpha * (nrSlice + 1))$$
$$z4 = radius2 * cos(\alpha * (nrSlice + 1))$$

Para representar corretamente o cone, é necessário dividir a altura do cone pelo número de camadas, o que nos vai permitir obter camadas iguais. Cada camada é definida por uma altura inferior, representada por h1 no código implementado, e por uma altura superior, representada por h2. Estas alturas vão corresponder aos y dos vértices dos triângulos formados. Tendo em conta o formato de um cone, é evidente que o raio diminui à medida que a altura aumenta, daí a necessidade de determinar os raios inferior e superior de cada camada.

Para obter fatias iguais, temos que determinar o ângulo α , dividindo 2π pelo número de fatias. Este ângulo é necessário para determinar as posições dos x e dos z de cada vértice dos triângulos.

Tal como já foi referido, o raio superior de cada camada é menor que o raio inferior, isto levou à necessidade de determinar quatro coordenadas x e quatro coordenadas z diferentes, correspondendo cada uma a cada vértice do espaço rodeado por uma camada e fatia. Tal não seria necessário se estivessemos a trabalhar com um cilindro, em que o raio é constante ao longo da altura. Nesse caso, apenas precisaríamos de determinar duas coordenadas x e duas coordenadas z.

Na primeira camada do cone é também formada a base dividida em fatias. Nas camadas interiores, o espaço contido entre cada camada e fatia é formado por dois triângulos, tal como se pode verificar na figura 5. A camada mais superior corresponde ao topo do cone, em que apenas é necessário um triângulo para dividir cada fatia.

O raciocínio desenvolvido foi aplicado ao número de camadas e de fatias desejados, tendo-se criado dois ciclos, sendo o ciclo exterior correspondente a cada camada do cone e o ciclo interior a cada fatia da camada. Em cada fatia, são calculados os vértices necessários para cada camada do cone. Assim, desenvolveu-se o seguinte algoritmo:

```
for(int j=1; j<=stacks; j++){
    h2 = tmp * j;
    radius2 = (height-h2) / tanB;
    for(int i=1; i<=slices+1; i++){
        x1 = r*sin(alpha*i);
        z1 = r*cos(alpha*i);
        x2 = r*sin(alpha*(i+1));
        z2 = r*cos(alpha*(i+1));
    x3 = radius2*sin(alpha*i);
        z3 = radius2*cos(alpha*i);
    x4 = radius2*sin(alpha*(i+1));
        z4 = radius2*cos(alpha*(i+1));

        if(j == 1){
            //base
            pointsList.push_back(new Point(0.0f,0,0.0f));
            pointsList.push_back(new Point(x2,0,z2));
            pointsList.push_back(new Point(x1,0,z1));

            //lados
            pointsList.push_back(new Point(x1,0,z1));
            pointsList.push_back(new Point(x2,0,z2));
            pointsList.push_back(new Point(x3,h2,z3));
            pointsList.push_back(new Point(x2,0,z2));
            pointsList.push_back(new Point(x4,h2,z4));
            pointsList.push_back(new Point(x3,h2,z3));
        }
        else if(j == stacks){
            pointsList.push_back(new Point(x1,h1,z1));
            pointsList.push_back(new Point(x2,h1,z2));
            pointsList.push_back(new Point(0,height,0));
        }
        else {
            //lados
            pointsList.push_back(new Point(x1, h1, z1));
            pointsList.push_back(new Point(x2, h1, z2));
            pointsList.push_back(new Point(x3, h2, z3));
            pointsList.push_back(new Point(x2, h1, z2));
            pointsList.push_back(new Point(x4, h2, z4));
            pointsList.push_back(new Point(x3, h2, z3));
        }
    }
    h1 = h2; r = radius2;}
```

2.2 Demonstração das figuras

2.2.1 Plano 3D

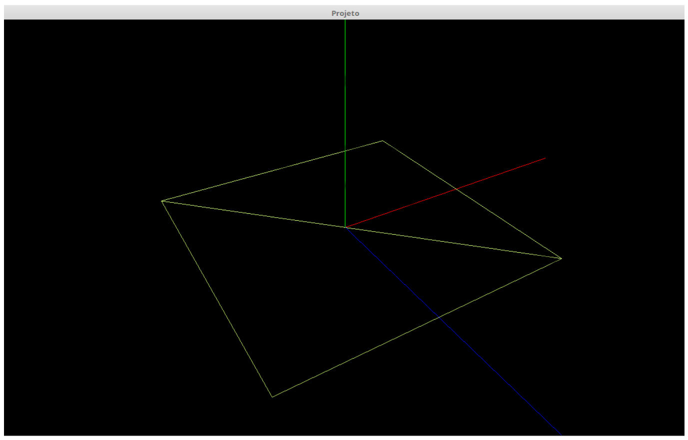

Figura 6: Plano

2.2.2 Caixa 3D

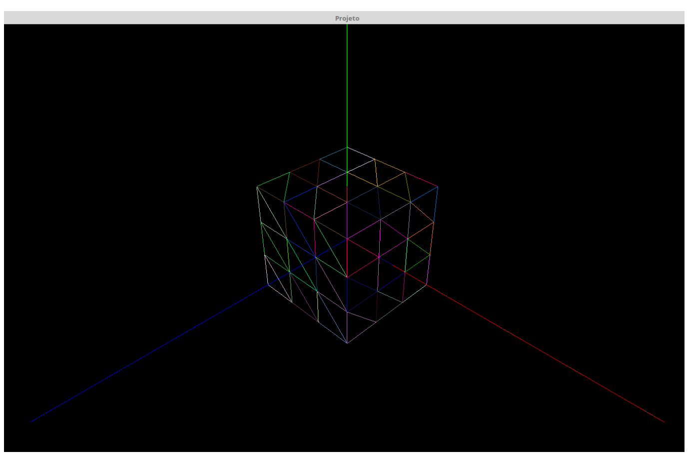

Figura 7: Caixa

2.2.3 Esfera 3D

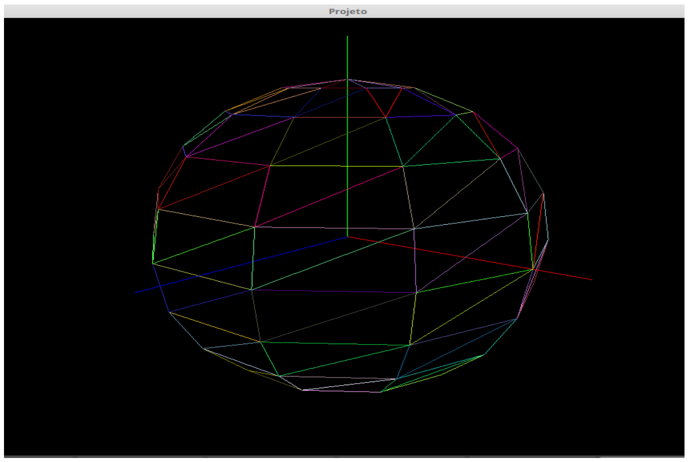

Figura 8: Esfera

2.2.4 Cone 3D

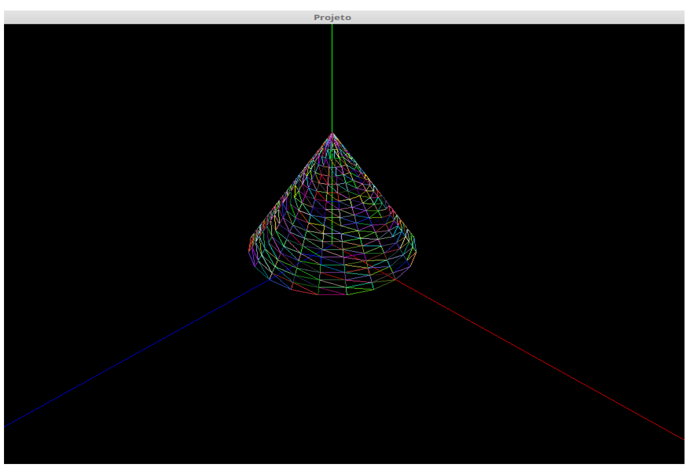

Figura 9: Cone

2.3 Motor

O motor, *engine*, tem como objetivo examinar informação no formato XML e desenhar em OpenGL as figuras contidas nos ficheiros encontrados no XML.

Para a análise do XML foi necessário uma classe que gerisse essa procura, a *Parser*. Além disso, foi oportuno utilizar a livraria *tinyxml2* que é capaz de explorar este tipo de documentos. Com essa livraria foi possivel abrir o ficheiro e efetuar uma análise sobre este, filtrando nomes de ficheiros nele contidos. De seguida, o motor é responsável por abrir os ficheiros encontrados e guardar os vértices incluidos nestes. Para tal, é necessário percorrer todas as linhas e capturar os vértices nelas presentes, ignorando a linha que separa as figuras. Desta forma, foi fundamental criar uma estrutura *vector<Point*>* que armazena todos os pontos encontrados. Ao capturar as três coordenadas de cada linha, é criado um objeto da classe *Point* e adicionado ao vector.

Após a conclusão da fase acima descrita, passa-se para a fase final onde se elabora o desenho das figuras através do OpenGL. Para tal fim, foi essencial percorrer os vértices contidos no *vector<Point*>* e evocar a função *glVertexf*.

Com o propósito de melhorar a visualização de cada triângulo projetado no ecrã, decidiu-se atribuir uma cor aleatória a cada um destes, sendo ainda possível visualizá-los no formáto sólido, linha ou ponto. Outras funcionalidades adicionais implementadas foram rotações, translações e movimento da câmera nos três eixos com o intuito de verificar a correta representação das figuras.

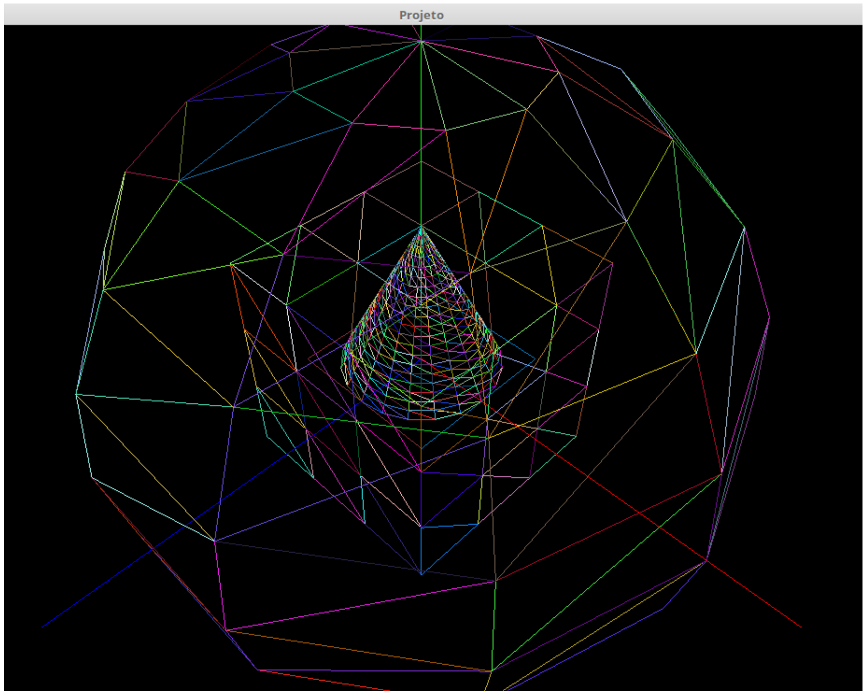

Figura 10: Exemplo de figura final

2.4 Execução das Aplicações

De forma a facilitar a utilização de cada aplicação, é fornecido um menu de ajuda no gerador e no motor. O menu descreve os comandos disponíveis dando um exemplo de utilização e explicitando qual o resultado esperado no final da aplicação.

3 Conclusões

A elaboração desta primeira fase permitiu-nos aprofundar os nossos conhecimentos em relação à ferramenta de coputação gráfica OpenGL e à linguagem de programação C++.

Todos os requisitos propostos para esta fase foram cumpridos, estando todas as funcionalidades implementadas e operacionais. Para além disso, considerou-se essencial o desenvolvimento de funcionalidades extras no motor para verificar a correta construção de todas as figuras. No entanto, encontrámos algumas dificudades na elaboração de certas figuras, nomeadamente a esfera e o cone, devido à necessidade de utilizar coordenadas polares e divisão por camadas.

O sucesso desta etapa será fundamental para a elaboração mais simplificada das seguintes fases do projeto.