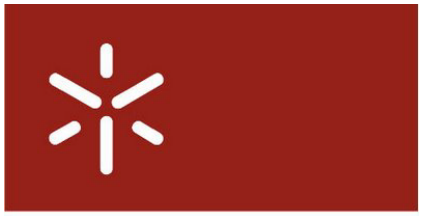


Universidade do Minho

Computação Gráfica

MIEI – 3º ano – 2º semestre

Universidade do Minho

Trabalho Prático - Parte 2

Nome	Nº. Mecanográfico
António Jorge Monteiro Chaves	A75870
Carlos José Gomes Campos	A74745
Cesário Miguel Pereira Perneta	A73883
Luís Miguel Bravo Ferraz	A70824

Introdução

No âmbito da Unidade Curricular de Computação Gráfica, numa primeira fase, foi-nos proposto, o desenvolvimento de um gerador de vértices de algumas primitivas gráficas (plano, caixa, esfera e cone).

Para além disto, também foi desenvolvido um mecanismo de leitura de ficheiros de configuração em XML2, que tem como objetivo desenhar os vértices das primitivas gráficas anteriormente geradas, a partir de ficheiros escolhidos pelo utilizador.

Na segunda fase, tivemos como objetivo o desenvolvimento de um gerador de vértices de algumas primitivas gráficas que nos permitem desenhar uma maquete do sistema solar, dos quais se destacam a esfera e o anel.

Para além disto, também foi desenvolvido um mecanismo de leitura de ficheiros de configuração em XML2, que para além de desenhar, dá cor, translação, rotação e escala às figuras.

Neste relatório incluímos uma descrição das várias etapas do trabalho, dando uma maior ênfase à descrição técnica do mesmo, com o recurso a equações, diagramas e figuras.

Fase 1: Primitivas Gráficas

1.1 Plano ZX

O desenho de um plano horizontal no eixo *zOx* é composto apenas por 2 triângulos. A chamada de geração de um plano requer a introdução de um número real, atribuído ao lado do plano. O valor calculado de $side = \frac{x}{2}$ permite-nos desenhar o plano com centro exato no meio do referencial, apenas variando os valores de x e z entre $[-side, side]$. De modo a que as figuras sejam visíveis pelo seu exterior, todos os triângulos devem ser desenhados tendo em conta a regra da mão direita.

```
 "plane":
try{
    x=(Double.parseDouble(args[1]))/2.0;
}catch(NumberFormatException e){
    System.out.println("Invalid Number");
}
sb.append(x+" 0 "+x+"\n");
sb.append(x+" 0 -"+x+"\n");
sb.append("-"+x+" 0 -"+x+"\n");
sb.append("-"+x+" 0 -"+x+"\n");
sb.append("-"+x+" 0 "+x+"\n");
sb.append(x+" 0 "+x+"\n");
```

Figura 1 Pontos do Plano

O desenho do plano na sua vista aérea (eixo *yy*) está representado no gráfico em baixo. Os pontos assinalados correspondem aos extremos do plano, com os quais serão formados os triângulos.

Para que o desenho seja visto desta perspetiva, a ordem pela qual os pontos dos triângulos devem ser escritos é:

- 1,2,3 Para o inferior;
- 2,4,3 Para o superior.

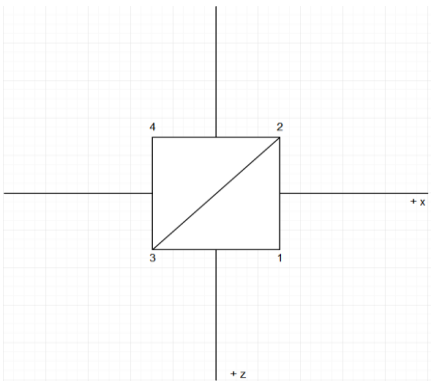


Figura 2 Gráfico do Plano

Os pontos são escritos linha a linha para um ficheiro cujo nome deve ser fornecido aquando da compilação do ficheiro. Esta operação é semelhante para todos casos de utilização do gerador.

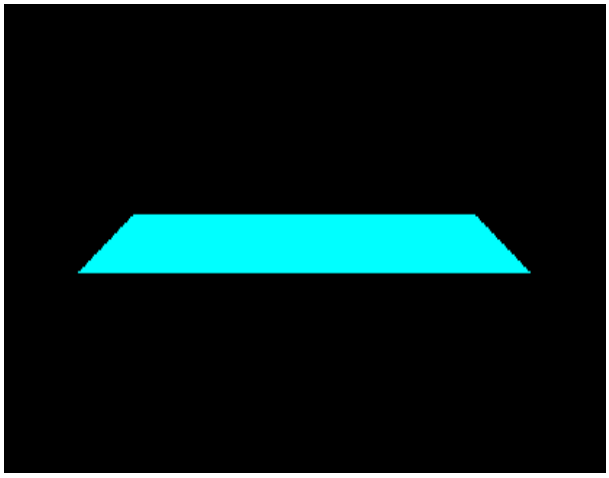


Figura 3 1ª Demo Scene Plano

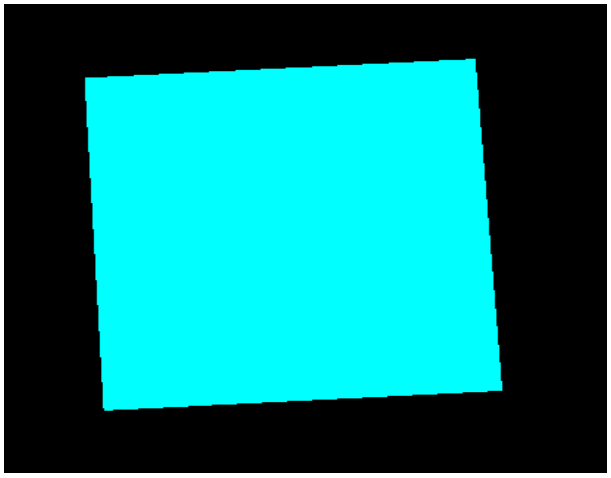


Figura 4 2ª Demo Scene Plano

1.2 Caixa

O desenho da caixa é dado por uma estratégia semelhante à do passo anterior. Uma caixa é composta por 6 planos, centrados na origem dos referenciais.

```
 "box":
try{
    x=(Double.parseDouble(args[1]))/2.0;
    y=(Double.parseDouble(args[2]))/2.0;
    z=(Double.parseDouble(args[3]))/2.0;
}catch(NumberFormatException e){
    System.out.println("Invalid Number");
}
// topo
sb.append(x+" "+y+" "+z+"\n");
sb.append(x+" "+y+" -"+z+"\n");
sb.append("-"+x+" "+y+" -"+z+"\n");
sb.append("-"+x+" "+y+" -"+z+"\n");
sb.append("-"+x+" "+y+" "+z+"\n");
sb.append(x+" "+y+" "+z+"\n");
// chao
sb.append(x+" -"+y+" "+z+"\n");
sb.append("-"+x+" -"+y+" "+z+"\n");
sb.append("-"+x+" -"+y+" -"+z+"\n");
sb.append("-"+x+" -"+y+" -"+z+"\n");
sb.append(x+" -"+y+" -"+z+"\n");
sb.append(x+" -"+y+" "+z+"\n");
// face esq
sb.append("-"+x+" -"+y+" "+z+"\n");
sb.append("-"+x+" "+y+" "+z+"\n");
sb.append("-"+x+" -"+y+" -"+z+"\n");
sb.append("-"+x+" -"+y+" -"+z+"\n");
sb.append("-"+x+" "+y+" "+z+"\n");
sb.append("-"+x+" "+y+" -"+z+"\n");
// face dir
sb.append(x+" -"+y+" -"+z+"\n");
sb.append(x+" "+y+" "+z+"\n");
sb.append(x+" -"+y+" "+z+"\n");
sb.append(x+" -"+y+" -"+z+"\n");
sb.append(x+" "+y+" -"+z+"\n");
sb.append(x+" "+y+" "+z+"\n");
// frente
sb.append(x+" -"+y+" "+z+"\n");
sb.append(" -"+x+" "+y+" "+z+"\n");
sb.append(" -"+x+" -"+y+" "+z+"\n");
sb.append(x+" -"+y+" "+z+"\n");
sb.append(x+" "+y+" "+z+"\n");
sb.append(" -"+x+" "+y+" "+z+"\n");
// tras
sb.append(x+" -"+y+" -"+z+"\n");
sb.append(" -"+x+" "+y+" -"+z+"\n");
sb.append(x+" "+y+" -"+z+"\n");
sb.append(x+" -"+y+" -"+z+"\n");
sb.append(" -"+x+" -"+y+" -"+z+"\n");
sb.append(" -"+x+" "+y+" -"+z+"\n");
```

Figura 5 Pontos Caixa

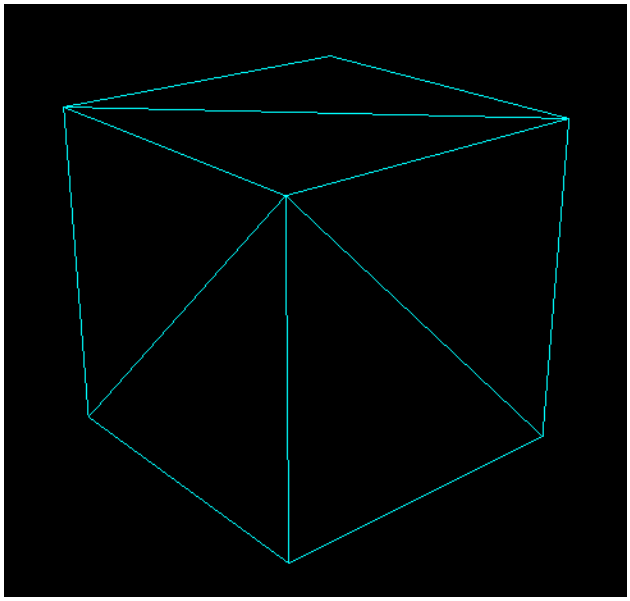


Figura 6 Demo Scene Caixa (GL_LINE)

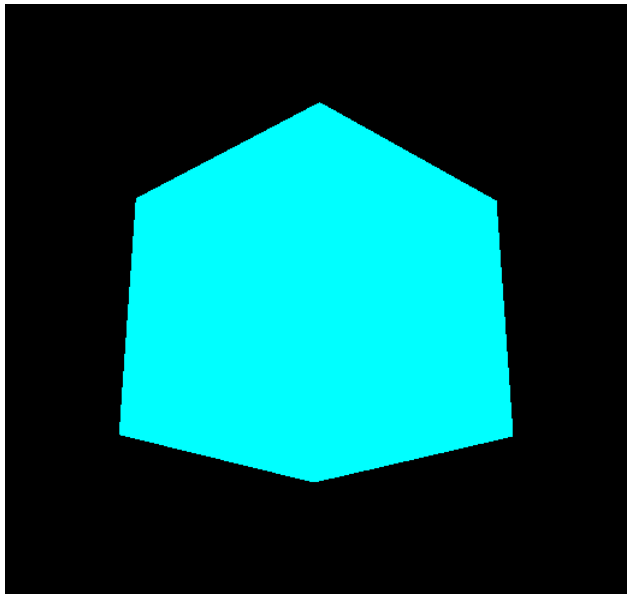


Figura 5 Demo Scene Caixa (GL_FILL)

Também é possivel definir a caixa como uma junção de várias caixas de tamanho inferior. Para esse efeito itera-se pelas variáveis e para cada ponto são calculados 12 triângulos. A figura 8 exemplifica o código de três faces de cada Caixa contida dentro da Caixa principal.

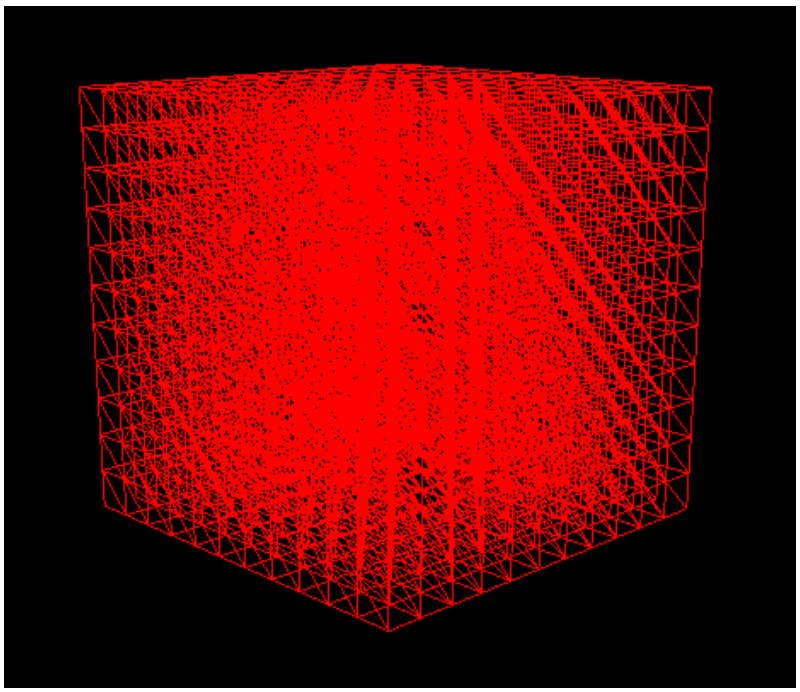


Figura 7 Demo Scene Caixa (iterativa)

- Division: Número de stacks e slices;
- varX: Comprimento do lado do cubo sobre o número de divisões pretendido;
- varY: Altura do cubo sobre o número de divisões pretendido;
- varZ: Comprimento do lado do cubo sobre o número de divisões pretendido;
- tmpx, tmpy, tmpz: Variáveis auxiliares para calcular as coordenadas do ponto seguinte.

Partindo de zero, as variáveis auxY, auxX e auxZ iteram até ao valor das coordenadas respetivas, sendo incrementadas por varY, varX e varZ respetivamente. Em cada iteração de z é desenhada uma linha de cubos, que em conjunto com as iterações de x dá origem ao desenho de uma stack. As iterações em y, por sua vez, indicam a altura de cada stack e o seu valor inicial de y, para que os triângulos sejam desenhados uniformemente e com extremidades concorrentes.

```
Double varX=x/division, varY=y/division, varZ=z/division;
Double tmpx=0.0, tmpy=0.0, tmpz=0.0;
for(Double auxY=0.0;auxY<=y;auxY+=varY)
    for(Double auxX=0.0;auxX<=x;auxX+=varX)
        for(Double auxZ=0.0;auxZ<=z;auxZ+=varZ){
            tmpx=(auxX+varX);
            tmpy=(auxY+varY);
            tmpz=(auxZ+varZ);
            // topo
            sb.append(tmpx+" "+tmpy+" "+tmpz+"\n");
            sb.append(tmpx+" "+tmpy+" "+auxZ+"\n");
            sb.append(auxX+" "+tmpy+" "+auxZ+"\n");
            sb.append(auxX+" "+tmpy+" "+auxZ+"\n");
            sb.append(auxX+" "+tmpy+" "+tmpz+"\n");
            sb.append(tmpx+" "+tmpy+" "+tmpz+"\n");
            // chao
            sb.append(tmpx+" "+auxY+" "+tmpz+"\n");
            sb.append(auxX+" "+auxY+" "+tmpz+"\n");
            sb.append(auxX+" "+auxY+" "+auxZ+"\n");
            sb.append(auxX+" "+auxY+" "+auxZ+"\n");
            sb.append(tmpx+" "+auxY+" "+auxZ+"\n");
            sb.append(tmpx+" "+auxY+" "+tmpz+"\n");
            // face esq
            sb.append(auxX+" "+auxY+" "+tmpz+"\n");
            sb.append(auxX+" "+tmpy+" "+tmpz+"\n");
            sb.append(auxX+" "+auxY+" "+auxZ+"\n");
            sb.append(auxX+" "+auxY+" "+auxZ+"\n");
            sb.append(auxX+" "+tmpy+" "+tmpz+"\n");
            sb.append(auxX+" "+tmpy+" "+auxZ+"\n");
```

Figura 8 Pontos Caixa

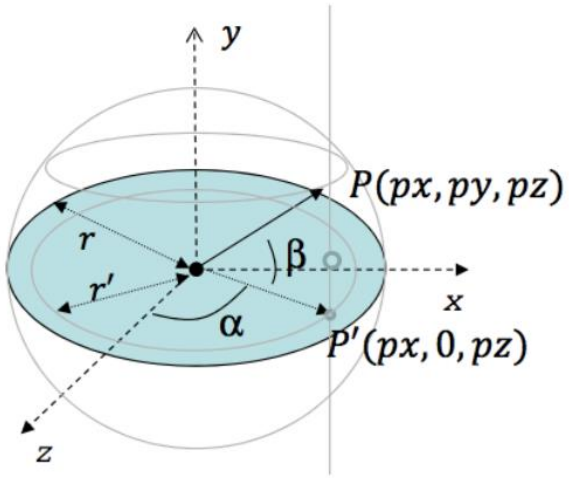
1.3 Esfera

O desenho da esfera requer a discriminação prévia de 3 variáveis, representando o raio da esfera, o número de *slices* horizontais e o número de *stacks* verticais, que definem o número de pontos a calcular. Para utilizar coordenadas esféricas no cálculo dos pontos de cada triângulo, definem-se 2 ângulos – *alfa* e *beta*, em radianos - e 2 incrementos angulares para cada iteração:

```
"sphere":
try{
    radius=Double.parseDouble(args[1]);
    slices=Double.parseDouble(args[2]);
    stacks=Double.parseDouble(args[3]);
}catch(NumberFormatException e){
    System.out.println("Invalid Number");
}
alfa=0.0; beta=-Math.PI/2; varAlfa=2*Math.PI/slices; varBeta=Math.PI/stacks;
for(double st=0.0;st<=stacks;st++){
    beta=-Math.PI/2 + varBeta*st;
    for(double sl=0;sl<=slices;sl++){
        alfa=varAlfa*sl;
        sb.append(radius*Math.cos(beta)*Math.sin(alfa)+" "+radius*Math.sin(beta)+" "+radius*Math.cos(beta)*Math.cos(alfa)+"\n");
        sb.append(radius*Math.cos(beta+varBeta)*Math.sin(alfa+varAlfa)+" "+radius*Math.sin(beta+varBeta)+" "+radius*Math.cos(beta+varBeta)*Math.cos(alfa+varAlfa)+"\n");
        sb.append(radius*Math.cos(beta+varBeta)*Math.sin(alfa)+" "+radius*Math.sin(beta+varBeta)+" "+radius*Math.cos(beta+varBeta)*Math.cos(alfa)+"\n");
        sb.append(radius*Math.cos(beta)*Math.sin(alfa)+" "+radius*Math.sin(beta)+" "+radius*Math.cos(beta)*Math.cos(alfa)+"\n");
        sb.append(radius*Math.cos(beta)*Math.sin(alfa+varAlfa)+" "+radius*Math.sin(beta)+" "+radius*Math.cos(beta)*Math.cos(alfa+varAlfa)+"\n");
        sb.append(radius*Math.cos(beta+varBeta)*Math.sin(alfa+varAlfa)+" "+radius*Math.sin(beta+varBeta)+" "+radius*Math.cos(beta+varBeta)*Math.cos(alfa+varAlfa)+"\n");
    }
}
```

Figura 9 Pontos Esfera

Para cada iteração, partindo de 0.0 e até ao número de slices dado, são calculados 4 pontos (P1, P2, P3 e P4 das páginas 9 e 10) e escritos para o ficheiro de texto indicado na compilação. Este ciclo é repetido para cada valor de beta, que percorre o diâmetro da esfera, ao longo do eixo yy.



1. $alfa = 0.0, alfa \in [0, 2\pi], varAlfa = \frac{2\pi}{slices}$
2. $beta = -\frac{\pi}{2}, beta \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right], varBeta = \frac{2\pi}{stacks}$

Figura 10 Coordenadas Esféricas

A descrição de um ponto no espaço, partindo de coordenadas esféricas para cartesianas segue a seguinte transformação:

$$(alfa, beta, r) => \begin{cases} z = r * \cos(beta) * \sin(alfa) \\ x = r * \cos(beta) * \cos(alfa) \\ y = r * \sin(beta) \end{cases}$$

Sendo assim, dados os 2 ângulos e o raio da esfera, podemos calcular as coordenadas cartesianas de qualquer ponto, com a vantagem de se poder calcular pontos próximos, apenas incrementando o ângulo *alfa* ou *beta*. Para cada *Stack*, *alfa* é inicializado a 0 e percorre os pontos de acordo com o incremento *varAlfa*. Para cada ponto, é preciso calcular as coordenadas de 4 pontos.

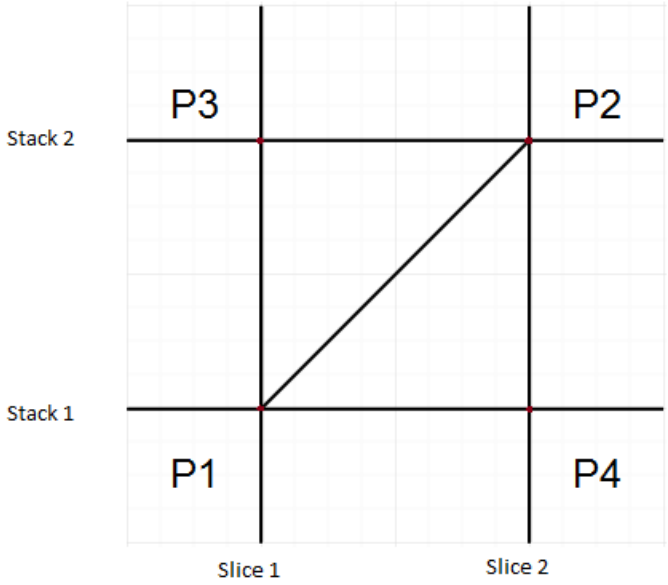


Figura 11 Pontos Triângulos

O primeiro ponto é o que coincide com a interseção da *slice* e *stack* iniciais de cada iteração, daí os valores das variáveis $alfa = alfa \ \& \ beta = beta$.

$$P1 => \begin{cases} z = r * \cos(beta) * \sin(alfa) \\ x = r * \cos(beta) * \cos(alfa) \\ y = r * \sin(beta) \end{cases}$$

O segundo ponto situa-se no vértice oposto ao primeiro, na interseção da *slice* e *stack* seguintes. O cálculo de (*x, y, x*) faz-se agora com $afla = alfa + varAlfa \ \& \ beta = beta + varBeta$.

$$P2 => \begin{cases} z = r * \cos(beta + varBeta) * \sin(alfa + varAlfa) \\ x = r * \cos(beta + varBeta) * \cos(alfa + varAlfa) \\ y = r * \sin(beta + varBeta) \end{cases}$$

Relativamente ao cálculo das coordenadas do terceiro ponto, pode observar-se que, em comparação com P1, se mantém o valor do ângulo horizontal *alfa*, enquanto que o valor de *beta* corresponde ao da próxima *stack*. O processo é análogo para P4, que no seu caso vê *alfa* a ser incrementado.

$$P3 => \begin{cases} z = r * \cos(beta + varBeta) * \sin(alfa) \\ x = r * \cos(beta + varBeta) * \cos(alfa) \\ y = r * \sin(beta + varBeta) \end{cases}$$

$$P4 => \begin{cases} z = r * \cos(beta) * \sin(alfa + varAlfa) \\ x = r * \cos(beta) * \cos(alfa + varAlfa) \\ y = r * \sin(beta) \end{cases}$$

Os 4 pontos calculados permitem desenhar 2 triângulos por ponto, por ordem P1, P2, P3 e P1, P4, P2. O programa gera para cada *stack* um "anel" de triângulos, ou seja, para cada iteração em *beta*, *alfa* percorre os pontos todos do anel.

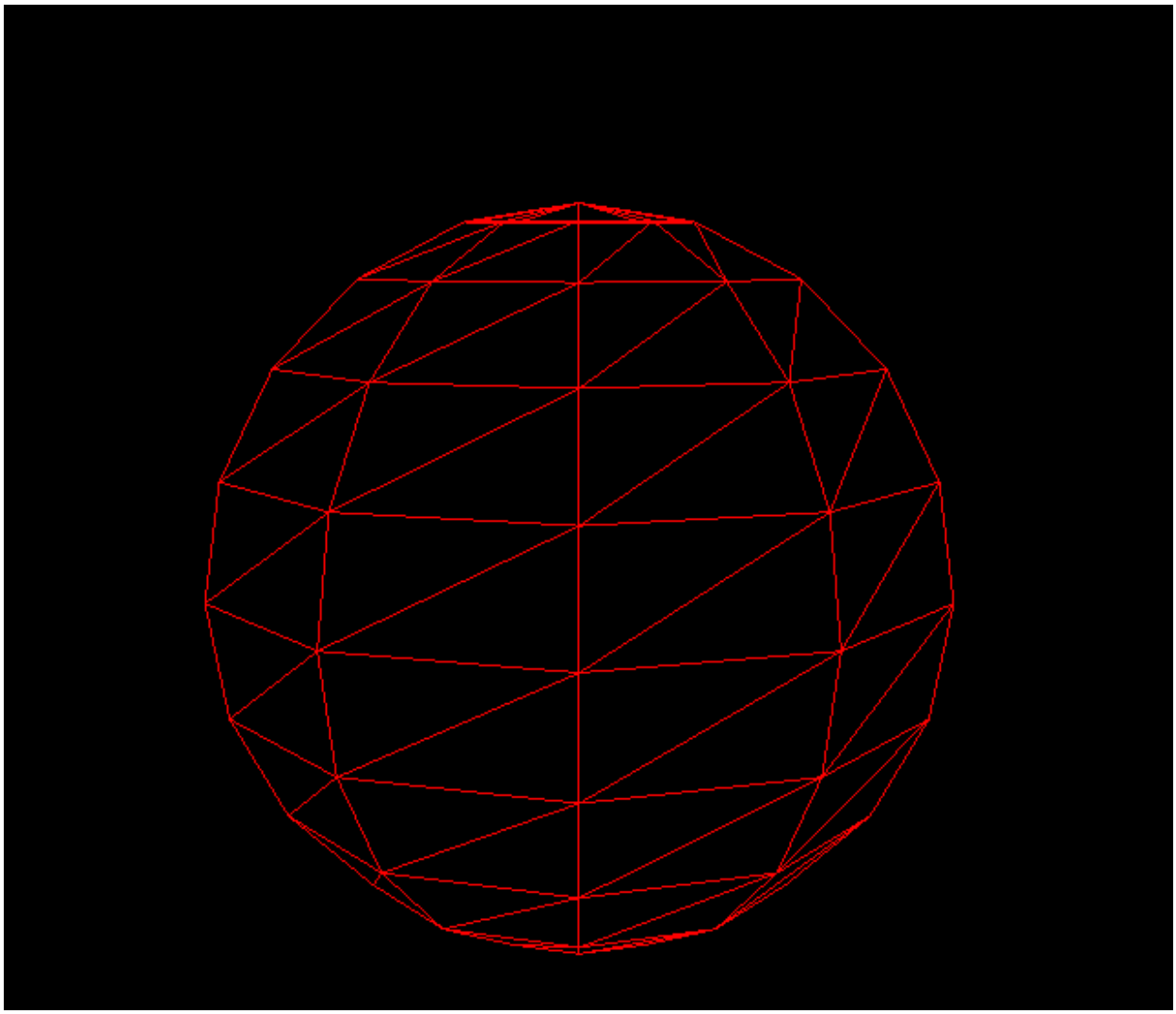


Figura 12 Demo Scene Esfera (GL_LINE)

1.4 Cone

Por último, o desenho do cone parte da transformação de variáveis polares em variáveis cartesianas, passando a altura do cone no parâmetro *y*. O conjunto de pontos calculados origina a face cónica do cone. Assim como no ponto anterior, a estrutura é percorrida gerando um anel de triângulos para cada *stack.* O valor da altura é fornecido aquando da compilação do ficheiro, junto com os valores do raio da base, número de *stacks* e número de *slices*.

```
"cone":
try{
    radius=Double.parseDouble(args[1]);
    height=Double.parseDouble(args[2]);
    slices=Double.parseDouble(args[3]);
    stacks=Double.parseDouble(args[4]);
}catch(NumberFormatException e){
    System.out.println("Invalid Number");
}
alfa=0.0;
double cheight=0.0;
double raio=radius;
double varHeight=height/stacks;
double varRadius=radius/stacks;
double varAngle=2*Math.PI/slices;
//topo
for(double ch=0;(ch<height&&raio>0);ch+=varHeight){
    for(double ca=0;ca<=2*Math.PI;ca+=varAngle){
        alfa+=ca*varAngle;
        sb.append((raio-varRadius)*Math.sin(ca)+" "+(cheight+varHeight)+" "+(raio-varRadius)*Math.cos(ca)+"\n");
        sb.append(raio*Math.sin(ca)+" "+(cheight)+" "+raio*Math.cos(ca)+"\n");
        sb.append((raio-varRadius)*Math.sin(ca+varAngle)+" "+(cheight+varHeight)+" "+(raio-varRadius)*Math.cos(ca+varAngle)+"\n");
        sb.append(raio*Math.sin(ca+varAngle)+" "+(cheight)+" "+raio*Math.cos(ca+varAngle)+"\n");
        sb.append((raio-varRadius)*Math.sin(ca+varAngle)+" "+(cheight+varHeight)+" "+(raio-varRadius)*Math.cos(ca+varAngle)+"\n");
        sb.append(raio*Math.sin(ca)+" "+(cheight)+" "+raio*Math.cos(ca)+"\n");
    }
    cheight+=varHeight;
    raio-=varRadius;
    alfa=0.0;
}
//base
double baseRad=radius;
double baseheight=0.0;
for(double baseAng=0;baseAng<2*Math.PI;baseAng+=varAngle){
    sb.append("0.0 0.0 0.0\n");
    sb.append(baseRad*Math.sin(baseAng+varAngle)+" "+baseheight+" "+baseRad*Math.cos(baseAng+varAngle)+"\n");
    sb.append(baseRad*Math.sin(baseAng)+" "+baseheight+" "+baseRad*Math.cos(baseAng)+"\n");
}
```

Figura 13 Pontos Cone

Para cada iteração de ch, variável que representa as stacks pelo incremento da altura em yy, são escritos, num ficheiro de texto, 3 coordenadas (1 ponto) por linha, referentes aos 4 diferentes pontos da face lateral do cone.

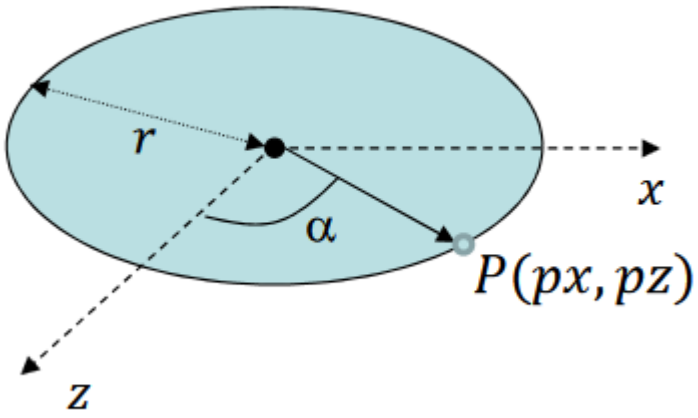


Figura 14 Coordenadas Polares

Os valores iniciais atribuídos às variáveis de altura e ângulo são os seguintes

$$height = 0.0, varHeight = height/stacks$$

$$alfa = 0.0, varAlfa = \frac{2\pi}{stacks}$$

A construção da primeira parte da figura calcula os pontos dos triângulos por uma ordem igual à das esferas, tendo em conta a transformação, neste caso, a partir de coordenadas polares.

$$(alfa, r) = \begin{cases} x = r * \cos(alfa) \\ y = height \\ z = r * \sin(alfa) \end{cases}$$

Os 4 pontos calculados para cada iteração de *alfa* são agora dados por:

$$P1 = \begin{cases} x = r * \cos(alfa) \\ y = height \\ z = r * \sin(alfa) \end{cases}$$

$$P2 = \begin{cases} x = (r - varRadius) * \cos(alfa + varAlfa) \\ y = height + varHeight \\ z = (r - varRadius) * \sin(alfa + varAlfa) \end{cases}$$

$$P3 = \begin{cases} x = (r - varRadius) * \cos(alfa) \\ y = height + varHeight \\ z = (r - varRadius) * \sin(alfa) \end{cases}$$

$$P4 = \begin{cases} x = r * \cos(alfa + varAlfa) \\ y = height \\ z = r * \sin(alfa + varAlfa) \end{cases}$$

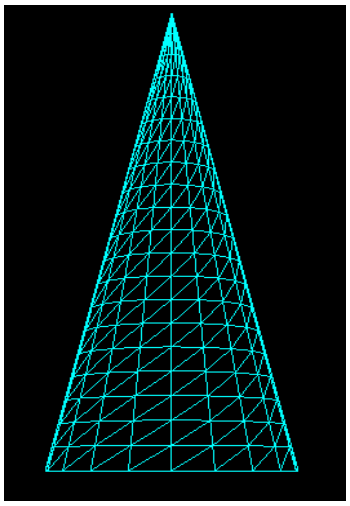


Figura 16 Demo Scene Cone

Observando o cone, pudemos afirmar que sempre que se pretende calcular um ponto que se encontra uma *stack* acima do original (pontos P2 e P3) é necessário incrementar a altura e decrementar o raio, ambos em proporção. O ângulo *alfa*, por sua vez, a cada ciclo da função oscila entre 0 e 2 π.

O desenho da base do cone faz-se partindo do centro da figura, juntando, para cada iteração, o ponto de ordem original, precedido do ponto de ordem seguinte.

$$P1 = \begin{cases} x = 0.0 \\ y = 0.0 \\ z = 0.0 \end{cases}$$

$$P2 = \begin{cases} x = r * \cos(alfa + varAlfa) \\ y = 0.0 \\ z = r * \sin(alfa + varAlfa) \end{cases}$$

$$P3 = \begin{cases} x = r * \cos(alfa) \\ y = 0.0 \\ z = r * \sin(alfa) \end{cases}$$

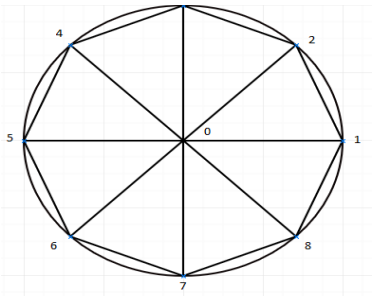


Figura 15 Exemplo de divisão do círculo

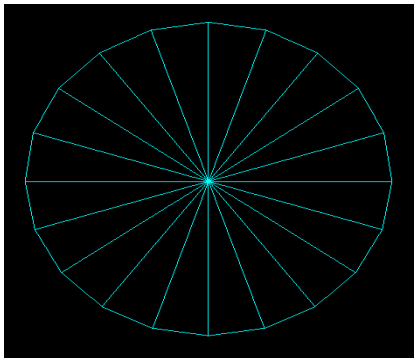


Figura 17 Exemplo de divisão do círculo

1.5 Motor

O ficheiro *Engine.h* tem como função a leitura dos pontos previamente gerados nos ficheiros ".3d". A função *xmlParser* começa por ler o ficheiro *config.xml* e retira a informação das figuras geométricas a desenhar. Para cada ficheiro, são lidas linha a linha as coordenadas *x, y e z*, e guardados numa instância da *class Point*, que por sua vez é adicionada a uma instância da *class Model*. No final da função o modelo é guardado num vetor invocado na função *processModels* que, para uma lista de modelos já compilados desenha os respetivos triângulos, através da sua chamada na função *renderScene*.

```
for(XMLElement *child=root->FirstChildElement("model");child!=NULL;child=child->NextSiblingElement()){
    const char * attrib=child->Attribute("file");
    std::ifstream xmlDoc(attrib);
    if(xmlDoc.is_open()){
        printf("XML File successfully opened.");
        /*Leitura dos pontos do ficheiro linha a linha, para o vector<Point>*/
        Model model;
        while(getline(xmlDoc,line)){
            std::stringstream stream(line);
            float px, py, pz;
            stream >> px >> py >> pz;
            Point newPoint(px,py,pz);
            model.points.push_back(newPoint);
        }
        xmlDoc.close();
        models.push_back(model);
        printf("XML Operation finished.");
    }
    else
        printf("Error opening XML file."+*xmlPath);
}
```

Figura 18 Ciclo xmlParser

A cada tag *<model>* lida, copia-se para uma string o nome do ficheiro *.3d* que é o argumento do atributo *file* de cada tag. O conjunto destas tags está sempre delimitado pelas tags <scene> </scene>.

```
1  <scene>
2      <model file="sphere.3d" />
3  </scene>
```

Figura 19 Exemplo de ficheiro de configuração

É aberta uma stream do ficheiro .3d cujos pontos já previamente escritos são lidos por uma *stringstream* para cada linha e separados pelas variáveis px, py e pz, antes de serem usados como parâmetros de uma nova instância da classe Point. Ao final da leitura do ficheiro, é procurada a próxima tag *<model* e o processo repete-se, para uma nova instância da class Model.

```
41  class Model{
42  public:
43      std::vector<Point> points;
44      void drawModel(){
45          glBegin(GL_TRIANGLES);
46          for(std::size_t i=0;i<points.size();i++){
47              glColor3f(1.0,0.0,0.0);
48              glVertex3f(points[i].getx(),points[i].gety(),points[i].getz());
49          }
50          glEnd();
51      }
52  };
53  std::vector<Model> models;
```

Figura 20 Classe Model e seu método drawModel

Para desenhar as figuras, é chamada a função drawModel de cada Model e chamada a função glVertex3f para o desenho de cada ponto, obtendo as suas coordenadas através de getx, gety e getz. A cada conjunto de 3 iterações da função é desenhado um triângulo.

Após o desenho das primitivas das figuras, foram inseridos Menus que permitem alterar a o preenchimento e cor das mesmas. Para aceder ao Menu basta apenas carregar na figura com o botão direito do rato.

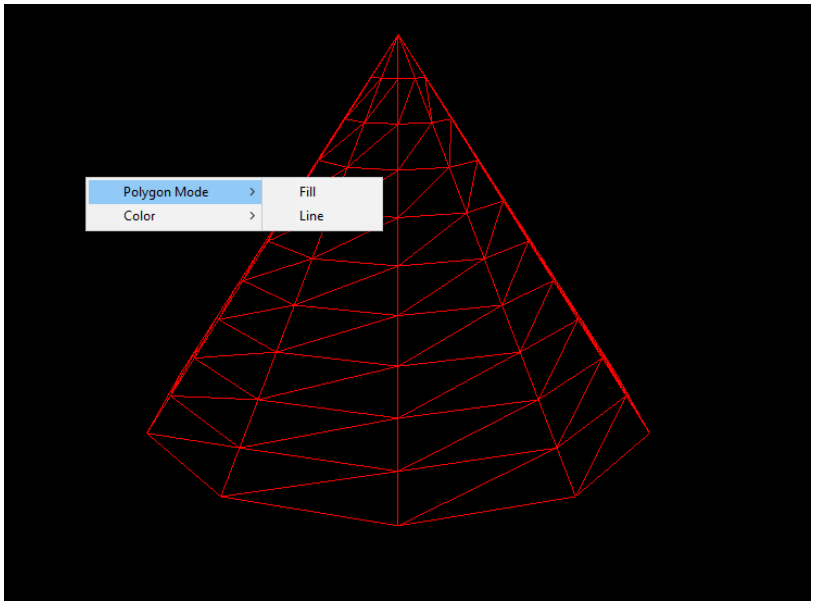


Figura 21 Exemplo Menu

```
float red = 1.0f, blue = 0.5f, green = 0.5f;
void processColorMenu(int option) {

    switch (option) {
    case RED:
        red = 1.0f;
        green = 0.0f;
        blue = 0.0f; break;
    case GREEN:
        red = 0.0f;
        green = 1.0f;
        blue = 0.0f; break;
    case BLUE:
        red = 0.0f;
        green = 0.0f;
        blue = 1.0f; break;
    case ORANGE:
        red = 1.0f;
        green = 0.5f;
        blue = 0.5f; break;
    }
}
```

Figura 23 Menu de Cor

```
void processFillMenu(int option) {

    switch (option) {

    case FILL: glPolygonMode(GL_FRONT, GL_FILL); break;
    case LINE: glPolygonMode(GL_FRONT, GL_LINE); break;
    }
}
```

Figura 22 Menu de tipo de desenho

Câmara

A câmara está posicionada com recurso a coordenadas polares nas variáveis camX, camY e camZ, sempre com a direção do centro do eixo referencial $(0,0,0)$. O vetor *up* coincide com um versor no eixo yy $(0,1,0)$.

```
float alfa = 0.0f, beta = 0.0f, radius = 20.0f, camX, camY, camZ;
float red = 1.0f, blue = 0.5f, green = 0.5f;
void spherical2Cartesian() {
    camX = radius * cos(beta)*sin(alfa);
    camY = radius * sin(beta);
    camZ = radius * cos(beta)*cos(alfa);
}
void renderScene(void) {
    // clear buffers
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // set the camera
    glLoadIdentity();
    glColor3f(red, green, blue);
    gluLookAt(camX, camY, camZ,
        0.0, 0.0, 0.0,
        0.0f, 1.0f, 0.0f);
    // drawing instructions
    processModels();
    // End of frame
    glutSwapBuffers();
}
```

Figura 24 Câmara

É possível alterar a posição da câmara com recurso às funções *glutSpecialFunc (processSpecialKeys)* e *glutKeyboardFunc (processNormalKeys)*.

```
void processKeys(unsigned char c, int xx, int yy) {
    if (c == 'f')
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    if (c == 'r')
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
}

void processSpecialKeys(int key, int xx, int yy) {
    switch (key) {
    case GLUT_KEY_RIGHT:
        alfa -= 0.1;
        break;
    case GLUT_KEY_LEFT:
        alfa += 0.1;
        break;
    case GLUT_KEY_UP:
        beta += 0.1f;
        if (beta>1.5f)
            beta = 1.5f;
        break;
    case GLUT_KEY_DOWN:
        beta -= 0.1f;
        if (beta<-1.5f)
            beta = -1.5f;
        break;
    case GLUT_KEY_PAGE_DOWN:
        radius -= 0.1f;
        if (radius<0.1f)
            radius = 0.1f;
        break;
    case GLUT_KEY_PAGE_UP:
        radius += 0.1f;
        break;
    }
    spherical2Cartesian();
    glutPostRedisplay();
}
```

Figura 25 Introdução por teclado

As teclas *f e r* alteram o preenchimento dos triângulos entre *LINE* e *FILL*, sendo que as seguintes são as teclas que alteram a posição da câmara:

- ←, →: incremento e decremento, respetivamente, em 0.1 unidades, do ângulo $alfa$.
- ↑, ↓: incremento e decremento, respetivamente, em 0.1 unidades, do ângulo $beta$. Este é limitado superior e inferiormente pelo módulo de 1.5.
- *PG_UP* e *PG_DOWN*: decremento e incremento, respetivamente da distância da câmara à origem, que não pode ultrapassar, no limite inferior, 0.1 unidades.

Fase 2: Transformações Geométricas

A proposta para a segunda etapa do trabalho prático era a inclusão de transformações geométricas na definição da cena, segundo um conjunto de transformações a aplicar a modelos dentro da mesma árvore.

```
<group name="Earth">
    <color R='0' G='0' B='255'/>
    <translation X='27'/>
    <scale X='1.7' Y='1.7' Z='1.7'/>
    <rotation Angle='7.2' X='1'/>
    <models>
        <model file="sphere.3d"/>
    </models>
    <group name="Moon">
        <color R='105' G='105' B='105'/>
        <translation X='1' Z='1'/>
        <scale X='0.2' Y='0.2' Z='0.2'/>
        <rotation Angle='5.2' X='1'/>
        <models>
            <model file="sphere.3d"/>
        </models>
    </group>
</group>
```

Figura 26 Exemplo de Transformações

São adicionadas 4 novas classes ao ficheiro engine.h, referentes aos tipos de alterações que podem ser feitas aos modelos, representados pelas tags XML *<color>, <translation>, <rotation>* e *<scale>*. Os parâmetros destas classes estão em conformidade com a assinatura das funções do *GLUT* $glTranslatef, glRotatef, glScalef$ e $glColor3f$.

```
class Translation {
public:
    float x, y, z;
    Translation::Translation(float xs, float ys, float zs) {
        x = xs; y = ys; z = zs;
    }
    Translation::Translation() {
        x = 0.0; y = 0.0; z = 0.0;
    }
};
class Scale {
public:
    float x, y, z;
    Scale::Scale(float xs, float ys, float zs) {
        x = xs; y = ys; z = zs;
    }
    Scale::Scale() {
        x = 0.0; y = 0.0; z = 0.0;
    }
};
```

```
class Rotation {
public:
    float angle,x, y, z;
    Rotation::Rotation(float ang,float xs, float ys, float zs) {
        angle = ang; x = xs; y = ys; z = zs;
    }
    Rotation::Rotation() {
        angle = 0.0; x = 0.0; y = 0.0; z = 0.0;
    }
};
class Color {
public:
    float r, g, b;
    Color::Color(float rs, float gs, float bs) {
        r = rs; g = gs; b = bs;
    }
    Color::Color() {
        r = 0.0; g = 0.0; b = 0.0;
    }
};
```

Figura 27 Classes de Transformações

A função *xmlParser* foi alterada para passar a chamar uma função auxiliar *parseGroup* a cada tag <group> no ramo imediato de <scene>. De seguida, itera-se pelas tags de cada classe e lê-se cada um dos seus parâmetros, armazenando-os em instâncias temporárias das respetivas. Na eventualidade de existência de *Groups* subsequentes, é corrida a função *parseGroup* recursivamente para esses grupos.

```
XMLElement *tmpRotation = child->FirstChildElement("rotation");
float rang = 0.0, rx = 0.0, ry = 0.0, rz = 0.0;
if (tmpRotation) {
    const char * tempang = tmpRotation->Attribute("angle");
    if (tempang) rang = atof(tempang);
    const char * tempx = tmpRotation->Attribute("X");
    if (tempx) rx = atof(tempx);
    const char * tempy = tmpRotation->Attribute("Y");
    if (tempy) ry = atof(tempy);
    const char * tempz = tmpRotation->Attribute("Z");
    if (tempz) rz = atof(tempz);
}
Rotation rot(rang, rx, ry, rz);
```

Figura 28 Leitura de parâmetros de Rotação

O funcionamento do código em relação às escalas, translações e cores é idêntico ao código das rotações que é ilustrado na imagem de cima. A classe Group é definida segundo a Figura 29.

```
for (XMLElement *tmpGroups = child->FirstChildElement("group"); tmpGroups != nullptr; tmpGroups = tmpGroups->NextSiblingElement()) {
    Group tmpG = parseGroup(tmpGroups);
    g_groupsTmp.push_back(tmpG);
}
Group gfinal(trans, rot, sca, col, g_modelsTmp, g_groupsTmp);
return gfinal;
```

Figura 29 Leitura de Groups subsequentes

```
class Group{
public:
    int type;
    Translation t;
    Rotation r;
    Scale s;
    Color c;
    std::vector<Model> g_models;
    std::vector<Group> g_groups;
    Group::Group(int tp,Translation ts,Rotation rs,Scale ss,Color cs,std::vector<Model> m,std::vector<Group> g){
        type=tp;t=ts;r=rs;s=ss;c=cs;g_models=m;g_groups=g;
    }
    Group::Group(){
        int typeg;
        Translation tg;
        Rotation rg;
        Scale sg;
        Color cg;
        std::vector<Model> mds;
        std::vector<Group> gps;
        t=tg;r=rg;s=sg;c=cg;g_models=mds;g_groups=gps;
    }
};
```

Figura 30 Classe Group

No final de cada iteração, a instância da classe Group gfinal é armazenada num vetor do tipo, junto com todos os conjuntos de figuras indicadas no ficheiro system.xml. Para o desenho da cena, percorre-se o vetor que contém todos os grupos, aplicando as transformações à matriz própria de cada Group. Todas as transformações de grupos subsequentes são aplicadas em relação à matriz do grupo "pai". A classe Group também possui um int, que serve para distinguir entre os planetas, luas e o sol da cintura de asteróides, que é gerada automaticamente, que irá ser demonstrada mais à frente.

```
void processGroup(Group g){
    if(g.type==1)
        drawBelt(g);
    else{
        glPushMatrix();
        glTranslatef(g.t.x,g.t.y,g.t.z);
        glRotatef(g.r.angle,g.r.x,g.r.y,g.r.z);
        glScalef(g.s.x,g.s.y,g.s.z);
        glColor3f(g.c.r,g.c.g,g.c.b);
        for(Model m:g.g_models)
            m.drawModel();
        for(Group gs:g.g_groups)
            processGroup(gs);
        glPopMatrix();
    }
}
```

Figura 31 Aplicação de Transformações

Cintura de Asteróides

```
void drawBelt(Group g){
    for(float i=0.0f;i<12.0f;i+=3.0f)
        for(float angle=0.0f;angle<2*M_PI;angle+=M_PI/36){
            glPushMatrix();
            glTranslatef(cos(angle)*(g.t.x+i),0,sin(angle)*(g.t.x+i));
            glColor3f(g.c.r,g.c.g,g.c.b);
            for(Model m:g.g_models)
                m.drawModel();
            glPopMatrix();
        }
}
```

Figure 32 Código que gera a cintura de asteróides

A cintura é gerada de forma automática e simples, o ciclo de dentro serve para desenhar um círculo de asteróides, fasados em PI/36 rad. O ciclo de fora desenha mais 3 círculos de asteróides, separados por 3 unidades de comprimento. Cada asteróide é nada mais do que uma esfera muito irregular, criados também pelo generator.java. O cálculo das translações de cada um é feito com ajuda do angulo, seguindo a lógica trignométrica convencional. A cor é dada de igual modo aos restantes elementos do sistema solar.

Gerador

Nesta fase do trabalho prático foi incluída a possibilidade de gerar os pontos de um anel através do ficheiro generator.java.

```
case "ring":
    try{
        radius=Double.parseDouble(args[1]);
        radius2=Double.parseDouble(args[2]);
        slices=Double.parseDouble(args[3]);
    }catch(NumberFormatException e){
        System.out.println("Invalid Number");
    }
    varAlfa=(2*Math.PI)/slices;
    for(alfa=0.0;alfa<2*Math.PI;alfa+=varAlfa){
        beta=alfa+varAlfa;
        sb.append(radius*Math.cos(beta)+" "+0.0+" "+radius*Math.sin(beta)+"\n");
        sb.append(radius*Math.cos(alfa)+" "+0.0+" "+radius*Math.sin(alfa)+"\n");
        sb.append(radius2*Math.cos(alfa)+" "+0.0+" "+radius2*Math.sin(alfa)+"\n");
        sb.append(radius2*Math.cos(alfa)+" "+0.0+" "+radius2*Math.sin(alfa)+"\n");
        sb.append(radius2*Math.cos(beta)+" "+0.0+" "+radius2*Math.sin(beta)+"\n");
        sb.append(radius*Math.cos(beta)+" "+0.0+" "+radius*Math.sin(beta)+"\n");
        sb.append(radius*Math.cos(beta)+" "+0.0+" "+radius*Math.sin(beta)+"\n");
        sb.append(radius2*Math.cos(beta)+" "+0.0+" "+radius2*Math.sin(beta)+"\n");
        sb.append(radius2*Math.cos(alfa)+" "+0.0+" "+radius2*Math.sin(alfa)+"\n");
        sb.append(radius2*Math.cos(alfa)+" "+0.0+" "+radius2*Math.sin(alfa)+"\n");
        sb.append(radius*Math.cos(alfa)+" "+0.0+" "+radius*Math.sin(alfa)+"\n");
        sb.append(radius*Math.cos(beta)+" "+0.0+" "+radius*Math.sin(beta)+"\n");
    }
```

Figura 33 Código de geração de anel

Para gerar um anel, é necessário fornecer os valores dos raios inferior e superior, assim como o número de divisões do anel. As coordenadas dos pontos de cada triângulo são calculadas recorrendo às coordenadas polares $(alfa, r) = \begin{cases} x = r * \cos(alfa) \\ y = 0.0 \\ z = r * \sin(alfa) \end{cases}$. O parâmetro de y=0.0 indica que os anéis serão sempre desenhados concorrentemente com o plano xOz. A cada iteração em torno do ângulo alfa são calculados 4 triângulos, agrupando 2 para desenho com vista superior e o resto com vista inferior. O cálculo dos pontos segue um raciocínio semelhante ao do cone.

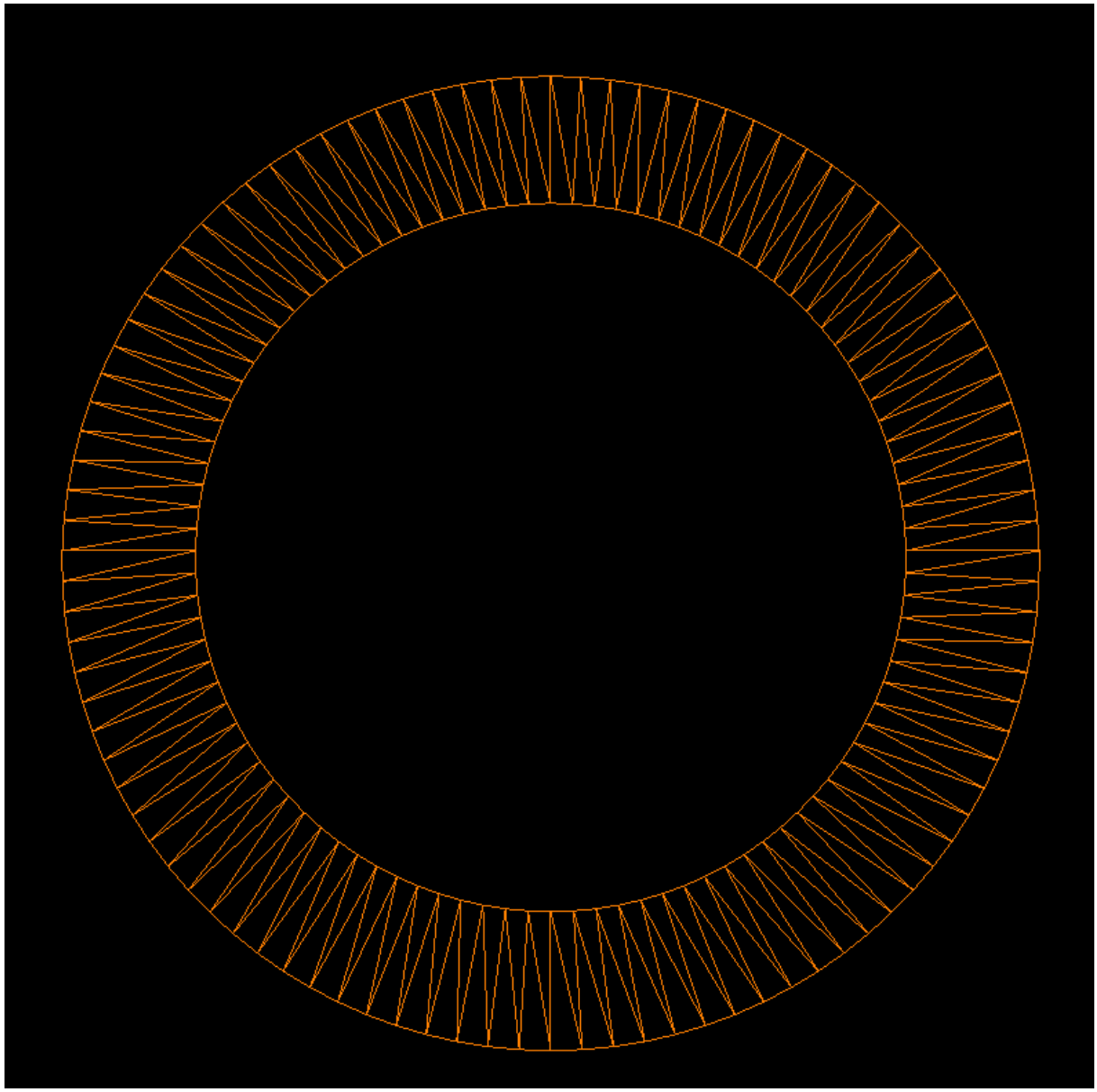


Figura 34 Demo Scene Anel (GL_FILL)

Câmera

A câmera nesta fase, continua a movimentar-se de maneira esférica, mas possui uma nova funcionalidade, que é poder alterar o centro de visão, ou seja, através de das teclas 'a', 'd', 'w', 's', 'q' e 'e', alterar os valores de origX, origY e origZ que são os valores que representam o ponto para o qual está a olhar, e alteram também os valores de camX, camY e CamZ, que são as coordenadas do ponto onde está posicionada a câmera. Em nota, salienta-se que houve uma alteação das teclas que alternam o preenchimento dos triângulos, para 'l' de *LINE* e 'f' de *FILL*. No fim da função processKeys é chamada a função gluLookAt para reposicionar a camara na nova posição.

```
void processKeys(unsigned char c,int xx,int yy){
    switch(c){
    case 'l':
        glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
        break;
    case 'f':
        glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
        break;
    case 'w':
        origY+=1.0f;
        camY+=1.0f;
        break;
    case 's':
        origY-=1.0f;
        camY-=1.0f;
        break;
    case 'a':
        origX-=1.0f;
        camX-=1.0f;
        break;
    case 'd':
        origX+=1.0f;
        camX+=1.0f;
        break;
    case 'q':
        origZ+=1.0f;
        camZ+=1.0f;
        break;
    case 'e':
        origZ-=1.0f;
        camZ-=1.0f;
        break;
    }
    glLoadIdentity();
    gluLookAt(camX,camY,camZ,origX,origY,origZ,0,1,0);
    glutPostRedisplay();
}
```

Figura 35 Código da nova câmera

Também foi adicionado a tecla HOME como o intuito de posicionar a câmera na posição original, reiniciando os parâmetros que influenciam a câmera.

```
void processSpecialKeys(int key,int xx,int yy){
    switch(key){
    case GLUT_KEY_RIGHT:
        alfa+=0.05f;
        break;
    case GLUT_KEY_LEFT:
        alfa-=0.05f;
        break;
    case GLUT_KEY_UP:
        beta+=0.05f;
        if(beta>1.5f)
            beta=1.5f;
        break;
    case GLUT_KEY_DOWN:
        beta-=0.05f;
        if(beta<-1.5f)
            beta=-1.5f;
        break;
    case GLUT_KEY_PAGE_DOWN:
        radius-=1.0f;
        if(radius<0.1f)
            radius=0.1f;
        break;
    case GLUT_KEY_PAGE_UP:
        radius+=1.0f;
        break;
    case GLUT_KEY_HOME:
        alfa=0.0f,beta=0.0f,radius=50.0f,
        camX=50.0f,camY=0.0f,camZ=0.0f;
        origX=0.0f,origY=0.0f,origZ=0.0f;
    }
    spherical2Cartesian();
    glLoadIdentity();
    gluLookAt(camX,camY,camZ,origX,origY,origZ,0,1,0);
    glutPostRedisplay();
}
```

Figura 36 Código da nova câmera

Demo Scene do Sistema Solar

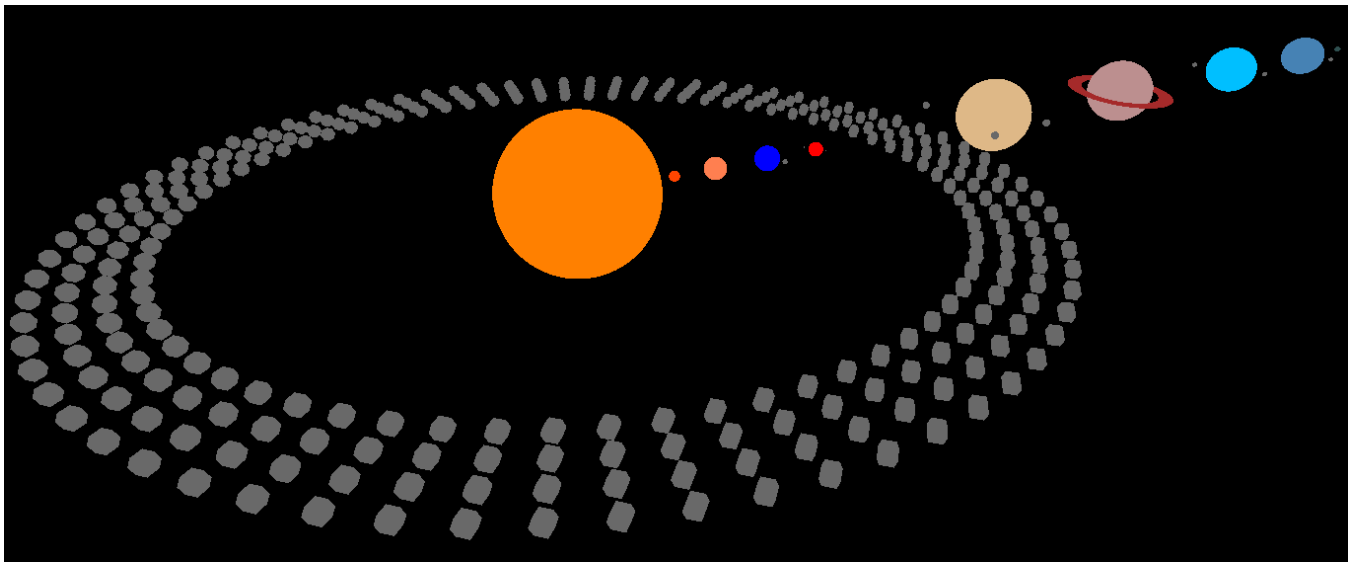


Figura 37 Demo Scene Fase 2 FILL

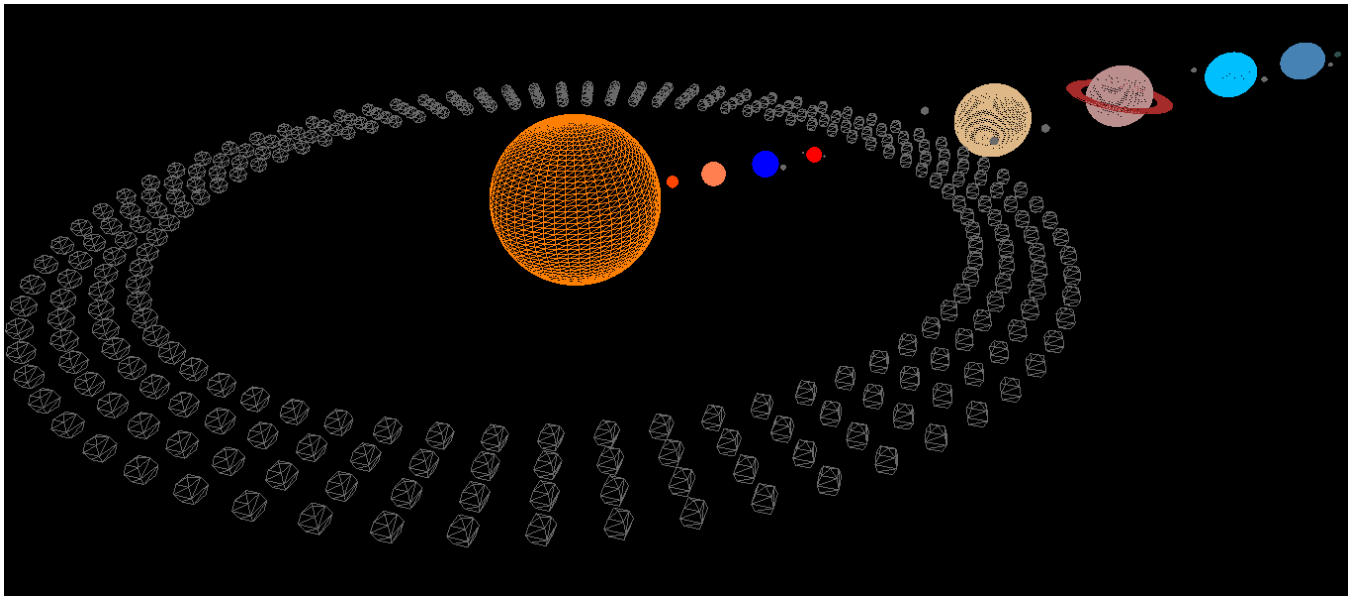


Figura 38 Demo Scene Fase 2 LINE