Computação Gráfica

Trabalho Prático

Fase 1 - Primitivas Gráficas

Luís Almeida	João Pedro Antunes	Fernando Lobo	Diogo Monteiro
A84180	A86813	A87988	A71452

13 de março de 2021

Resumo

O presente documento descreve a resolução do enunciado adotada pelo grupo. Foram desenvolvidas duas aplicações, um *generator*, que gera um ficheiro contendo os vértices e as faces de uma primitiva gráfica especificada pelo utilizador, assim como um *engine*, que lê um ficheiro *XML* contendo o nome dos ficheiros gerados pelo *generator* para posterior carregamento e renderização dos vértices e faces contidas nos mesmos.

Conteúdo

Capítulo 1

Introdução

Foi nos proposta a criação de um *generator* e de um *engine* que façam a geração de pontos e faces de primitivas gráficas assim como a renderização dos mesmos, respetivamente. Ao longo deste relatório iremos descrever com detalhe os diversos problemas encontrados, as soluções adotadas pelo grupo assim como uma opinião crítica do trabalho realizado.

Capítulo 2

Análise e Especificação

Atendendo ao problema descrito na introdução, apresentam-se os requisitos pedidos:

É necessário escrever duas aplicações, um *generator* e um *engine*. O *generator* terá de ser capaz de gerar os pontos e faces das seguintes primitivas gráficas:

1. Plano (tamanho do lado especificado pelo utilizador)
2. Cubo (parametrizado por dimensões X,Y,Z)
3. Esfera (parametrizada por raio, slices e stacks)
4. Cone (parametrizado por raio da base, altura, slices e stacks)

O *generator* terá de receber informação sobre os modelos a serem gerados (por exemplo tamanho do lado) especificada pelo utilizador e criar a primitiva correspondente, gravando a informação gerada num ficheiro também ele indicado pelo utilizador.

O *engine* terá de receber um ficheiro *XML* com os nomes dos ficheiros gerados pelo *generator* a carregar para memória para posterior renderização dos mesmos.

Capítulo 3

Concepção/desenho da Resolução

3.1 *Shape Generator*

Nesta secção trataremos do desenho conceptual e optimizações adotadas para a geração dos *shapes* do gerador, desde os cálculos para a determinação de seus vértices em um espaço 3D até sua escrita e representação em ficheiro.

3.1.1 Representação por Índices

Para a definição de Sólidos em um espaço 3D utilizando-se das ferramentas apresentadas do *OpenGL* e *Glut* precisamos definir a posição dos vérices no espaço 3D, isto é suas coordenadas nos eixos (x,y,z), e também a ordem de renderização pela qual o *OpenGL* deve se guiar para realizar o *Cull* destas faces.

As soluções naturais deste problema, para o grupo, foram duas: armazenar os vértices pela ordem de *Cull* ou associarmos a cada vértice um respetivo índice e de seguida organizar os índices na correta ordem de *Cull*.

A escolha, como indica o nome desta secção, deu-se à associação de índices. Foi dada por o grupo acreditar ser a melhor maneira de armazenar os pontos em memória, evitando repetições e poupando erros de representação.Além disso, a mesma facilitou, como visto mais adiante a implementação de *VBOs* na Engine.

A aplicação da ideia consistia então de 3 grandes procedimentos:

1. Gerar Vértices;
2. Atribuir índices;
3. Gerar Triângulos;

Uma maneira de solucionar a questão foi criar uma depêndencia entre a atribuição de índices e a geração de triângulos, assim, o problema apresentado reduziu-se a um problema de **planificação**, garantindo um *Cull* uniforme das faces como apresentado na próxima secção.

3.1.2 Planificação

Dando continuidade a ideia inicial de geração de vértices e atribuição de índices para a formação de triângulos, a técnica de planificar o objeto foi útil para facilitar a geração de figuras dinâmicamente, sem a necessidade de especificar algoritmos de criação de geração de triângulos para cada shape desejado.

Assim, consideramos um algoritmo base de geração de triângulos baseado em uma *Mesh* (ou malha) definida por uma Matriz, cujas entradas são os índices dos vértices a representar no shape. Como exemplo define-se abaixo o processo de criação de triângulos a partir de uma malha:

Tomemos a seguinte matriz como exemplo,

$$\begin{bmatrix} a & b & -1 & c & d & -1 \\ e & f & -1 & g & -1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

Se analisarmos a matriz como uma malha, considerando que (a,b,c,d,e,f,g) são índices de vértices e que não hajam triângulos com vértices de índice não positivo.

Podemos, então, tomar uma métrica constante para criação de triângulos, por exemplo, apenas triângulos superiores esquerdos. Teríamos como resultado uma lista de vértices de triângulos, com o *Cull* ajustado para que a malha estivesse uniforme e apontada sempre para a mesma direção, gerando a superfície desejada. Na matriz apresentada, teríamos, então, 3 triângulos superiores esquerdos:

1. *(e,b,a)*;
2. *(e,f,b)*;
3. *(g,d,c)*;

Outra ideia para se verificar o algoritmo seria analisar toda matriz de índices como se sobreposta ao seguinte diagrama:

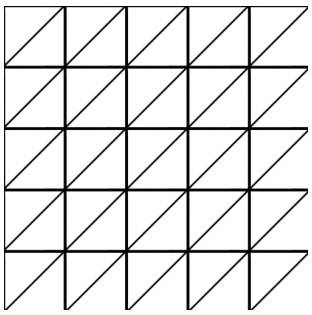

Figura 3.1: Diagrama da Malha

Em que vértice de índice não positivo constituem a não existência de uma edge. Os triângulos ecolhidos e a direção do *Cull* não importam tanto para esta etapa do algoritmo, aqui, a importância dá-se à uniformidade do *Cull*, i.e. estarem todas as faces a apontar para o mesmo lado. A seguir demonstra-se uma maneira de realizar o processo descrito.

```
for (int i = 1; i < lins; i++){
        for (int h = 0; h < cols; h++){
                int p = (h+1)%(cols);

                if ((planar[i][h] > 0 && planar[i-1][h]> 0 && planar[i-1][p]> 0)    &&  (
                    planar[i][h] != planar[i-1][h]) && (planar[i][h] != planar[i-1][p]) &&
                     (planar[i-1][h]!= planar[i-1][p])){
                        faces.push_back(planar[i][h]);
                        faces.push_back(planar[i-1][h]);
                        faces.push_back(planar[i-1][p]);
                        t++;
                }
                if ((planar[i][h] > 0 && planar[i-1][p] > 0 && planar[i][p] > 0)   &&  (
                    planar[i][h] != planar[i-1][p]) && (planar[i][h] != planar[i][p]) && (
                    planar[i-1][p]!= planar[i][p])){
                        faces.push_back(planar[i][h]);
                        faces.push_back(planar[i-1][p]);
                        faces.push_back(planar[i][p]);
                        t++;
                }
        }
}
```

Em que "planar"representa uma matriz nas condições descritas. A seguir, trataremos a atribuição de índices de vértices a uma malha.

3.1.3 Relação Vértices ⇒ Planificação

A etapa seguinte consiste, basicamente de "embrulhar"a malha de triângulos descrita, no shape desejado, posicionando os vértices e seus devidos índices em uma matriz, que deverá ser passada pelo tratamento apresentado.

Neste ponto cada *Shape* terá seu próprio procedimento, para a devida definição de vértices e sua planificação.

Sphere

Para o *assembly* da esfera, tomaremos proveito das coordenadas esféricas em conjunto com o padrão bidimensional da matriz de planificação definida anteriormente.

O processo tomará em conta que a associação de vértices com índices pode ser feita trivialmente, a cada novo vértice, o índice é sucessor do último vértice gerado. Para associação de vértices com sua correspondência na matriz, fará uma associação entre linhas horizontais (resp. verticais) na matriz, com as *stacks* (resp. *slices*).

Assim a posição de um dado vértice pode ser dada através de coordenadas esféricas em conjunto com a matriz de planificação. Gerando, então, todos os requisitos do *shape*, a seguir demonstra-se o código respetivo:

```
void shape(){
    float freqalf = (2*M_PI)/slices;
    float freqbet = (M_PI)/(stacks+1);
    int vtxPts = 1;
    std::vector<int> row;

    for (int st = 0; st < stacks+2; st++){
        row.clear();
                for (int sl = 0; sl < slices; sl++){
                    if (!(st == 0 || st == stacks+1) || sl == slices-1){
                                std::vector<float> aux;
                                aux.push_back((cos(freqbet*st - (M_PI/2))*radius)*sin(
                                    freqalf*sl));
                                aux.push_back((float)radius*sin(freqbet*st - (M_PI/2)));
                                aux.push_back((cos(freqbet*st - (M_PI/2))*radius)*cos(
                                    freqalf*sl));
                                (this->vertex).push_back(aux);
                                row.push_back(vtxPts++);
                        }
                    else row.push_back(vtxPts);
                }
                (this->planar).push_back(row);
        }
}
```

Cone

O cone seguirá um processo muito semelhante ao apresentado na esfera, teremos a mesma relação entre *stacks* e *slices* com a matriz de planificação. A maior diferença para a esfera será que aqui não utilizaremos o sistema de coordenadas esférico, mas sim um sistema de coordenadas cilíndrico. Moldando os vértices devidamente ao formato desejado.
A seguir, o código:

```
void shape(){
        int vtxPts = 1;
        float rat = (radius)/stacks;
        float heiStck = height/(stacks);
        float freq = (2*M_PI)/slices;
        int inv;
        std::vector<int> row;
        std::vector<float> aux;
        for (int st = 0; st <= stacks+1; st++){
        row.clear();
                for (int sl = 0; sl < slices; sl++){
                        if (!(st == 0 || st == stacks+1) || sl == slices-1){
                                aux.clear();
                                inv = (stacks+1 - st)%(stacks+1);
                                aux.push_back((rat*inv)*sin(freq*sl));
                                aux.push_back((-(height))/2 +(heiStck)*(st-1));
                                aux.push_back((rat*inv)*cos(freq*sl));
                                if (st == 0)aux[1] =(-(height))/2;
                (this->vertex).push_back(aux);
                                row.push_back(vtxPts++);
                        }

                        else row.push_back(vtxPts);
                }
                (this->planar).push_back(row);
        }
}
```

Plane

O plano será o *shape* mais trivial para o sistema escolhido, basta utilizarmos as coordenadas na malha com as especificações de dimensões requeridas na geração do plano para termos as posições dos vértices orientadas à malha desejada.

```
void shape(){
        int vtxPts = 1;
        float xRatio = xDim/xSlc;
        float zRatio = zDim/zSlc;

        std::vector<float> rots;
        std::vector<float> aux;
        std::vector<int> row;

        for (int x = 0;  x < xSlc+1; x++){
                row.clear();
                for (int z = 0; z < zSlc+1; z++){
                        aux.clear();
                        rots.clear();
                        //gera o ponto
                        aux.push_back((-(xDim)/2) + x*xRatio);
                        aux.push_back(0);
                        aux.push_back((-(zDim)/2) + z*zRatio);
                        //rotate xAx
                        rots.push_back(aux[0]);
                        rots.push_back(-aux[2]*sin(xAng));
                        rots.push_back(aux[2]*cos(xAng));
                        //rotate zAx
                        aux[0] = rots[0]*cos(zAng) - rots[1]*sin(zAng);
                        aux[1] = rots[0]*sin(zAng) + rots[1]*cos(zAng);
                        aux[2] = rots[2];
                        //translada para centrar na origem desejada
```

```
                        aux[0] += orig[0];
                        aux[1] += orig[1];
                        aux[2] += orig[2];

                        (this->vertex).push_back(aux);
                        row.push_back(vtxPts++);
                }
                row.push_back(-1);
                (this->planar).push_back(row);
        }
}
```

Nota-se, entretanto, no excerto de código apresentado a presença de outros critérios para a determinação dos pontos, tais como rotações. Estes parâmentros, apesar de não úteis ao plano serão de grande importância para o sólido da caixa, e não apresentam nenhuma perda se ignorados para a criação de um plano simples.

Box

A caixa é, nesse sistema, a maior complicação, uma vez que não apresenta uma planificação trivial com relações matemáticas simples.

A maneira de contornar o problema utilizada foi a geração de 6 planos, utilizando-se dos parâmentros passados a caixa para a criação de planos com as propriedades desejadas. Rodando e deslocando os planos gerados, conforme a necessidade. E por fim, unificando os planos gerados em um único polígono.

```
void shape(){
        planeFaces.push_back(Plane(yDim,zDim,slicesY,slicesZ,"",{xDim/2,0,0},0,-M_PI/2));
        planeFaces.push_back(Plane(yDim,zDim,slicesY,slicesZ,"",{-xDim/2,0,0},0,M_PI/2));

        planeFaces.push_back(Plane(xDim,zDim,slicesX,slicesZ,"",{0,yDim/2,0},0,0));
        planeFaces.push_back(Plane(xDim,zDim,slicesX,slicesZ,"",{0,-yDim/2,0},0,M_PI));

        planeFaces.push_back(Plane(xDim,yDim,slicesX,slicesY,"",{0,0,zDim/2},M_PI/2,0));
        planeFaces.push_back(Plane(xDim,yDim,slicesX,slicesY,"",{0,0,-zDim/2},-M_PI/2,0));

        this->unifyFaces();
}
```

3.1.4 Representação em ficheiro

Por fim, tendo todos os devidos *shapes* funcionais, a escrita para ficheiro escolhida segue um modelo simples, em que o ficheiro é escrito em texto. Inicialmente escreve-se o número de vértices do sólido, seguido de uma nova linha com o número de faces.

De seguida, escrevem-se todos os vértices seguidos, com suas componentes (x,y,z) separadas por linhas. Por fim, são escritos os índices de cada vértice que compõe as faces, pela ordem correta de *Cull* do gerador.

```
int write3DFile(){
        std::ofstream File2Wr;
        File2Wr.open(this->fileName);
        File2Wr << vex << "\n" << facs <<"\n";
        for (std::vector<float> &v : this->vertex){
            for (float &point : v){
                        File2Wr << point << "\n";
                }
        }
        for (int face:faces){
                File2Wr << face << "\n";
        }
        File2Wr.close();
}
```

3.2 Engine

3.2.1 Parsing *XML*

Conforme referido no capítulo anterior esta aplicação terá de ler um ficheiro *XML* que irá conter o nome dos ficheiros com informação sobre os vértices e faces dos modelos especificados. Sendo assim, o primeiro passo é fazer o *parsing* do ficheiro *XML* e extrair o nome dos ficheiros a carregar. Para este efeito usamos o *tinyXML2*, que nos permite iterar sobre um ficheiro *XML* e extrair a informação que nos é relevante de uma forma simples. Abrimos o ficheiro *XML* especificado aquando a chamada do programa e iteramos o mesmo da seguinte forma:

```
XMLElement* iterator;
for(iterator = doc.FirstChildElement()->FirstChildElement();iterator != NULL; iterator =
    iterator->NextSiblingElement()){

                const char* file_name = iterator->Attribute("file");
                listXML.push_back(std::string(file_name));
        }
```

Para armazenarmos em memória todos estes nomes de ficheiros criamos um vetor *listXML* que irá conter esta informação.

3.2.2 Carregamento dos Modelos

Para mantermos em memória central todos os vértices e faces de todos os modelos a serem renderizados torna-se relevante criar uma classe *Model* que irá conter a informação específica a cada um deles, tendo presente como variável global um vetor de *Model*:

```
class Model{
        public:

        std::vector<float> vertexList;
        std::vector<unsigned int> facesList;
        unsigned int indexNumber;
        GLuint vertex;
        GLuint indices;
}
```

O carregamento em memória dos modelos criados pelo *generator* é feito após o *parsing* do ficheiro *XML*. Iteramos o vetor *listXML*, lendo os vértices e faces presentes em cada ficheiro. Tendo em conta a estrutura do ficheiro 3d especificada anteriormente, desenvolvemos o seguinte método que carrega para o objeto *Model* a informação contida num ficheiro:

```
      void readFile( std::string fileName){
                std::ifstream mod;
                mod.open(fileName.c_str());

                int numVertices;
                int numFaces;
                float auxf;
                unsigned int auxi;

                mod >> numVertices;
                mod >> numFaces;

                for(int i = 0; i < 3*numVertices;i++){
                        mod >> auxf;
                        vertexList.push_back(auxf);
                }
```

```
                for(int i = 0; i < 3*numFaces; i++){
                        mod >> auxi;
                        facesList.push_back(auxi-1);
                }
                mod.close();
        }
```

Após esta fase temos então um vetor com a informação acerca de cada modelo em memória.

3.2.3 Renderização

Queremos agora desenhar os modelos carregados. A maneira mais simples seria iterar sobre o vetor de *Model* e desenhar cada um dos pontos presentes na *vertexList* de cada modelo consoante a ordem especificada na *facesList*. Assim, desenvolvemos o seguinte método da classe *Model*:

```
void drawT(){
  glBegin(GL_TRIANGLES);
  for(unsigned int i: facesList){
  glVertex3f(vertexList[(3*i)],vertexList[3*i+1],vertexList[3*i+2]);
  }
  glEnd();
}
```

Ora, basta então na função *renderScene* iterar o vetor de *Model* e invocar o método *drawT* para cada objeto encontrado.

3.2.4 VBOs

No entanto, para podermos tirar partido do *OpenGL* e fazermos otimizações, podemos utilizar *Vertex Buffer Objects* para armazenar em VRAM os pontos e a ordem em que serão desenhados. Torna-se então necessária a preparação da informação para ser usada neste contexto, pelo que definimos o seguinte método:

```
void prepareDataVBO(){
        glGenBuffers(1,&vertex);
        glBindBuffer(GL_ARRAY_BUFFER, vertex);
        glBufferData(GL_ARRAY_BUFFER,(unsigned int)((sizeof(float))*vertexList.size()),
            vertexList.data(), GL_STATIC_DRAW);

        glGenBuffers(1,&indices);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indices);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER,(unsigned int)((sizeof(unsigned int))*
            facesList.size()), facesList.data(), GL_STATIC_DRAW);

        glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);

    indexNumber = (unsigned int) facesList.size();
}
```

O método *prepareDataVBO* terá de ser invocado apenas uma vez após a leitura do ficheiro 3d. Posto isto, definimos um método que desenha os VBOs, que será chamado pela *renderScene*:

```
void drawVBO(){
        glBindBuffer(GL_ARRAY_BUFFER,vertex);
        glVertexPointer(3, GL_FLOAT, 0, 0);

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indices);
        glDrawElements(GL_TRIANGLES, indexNumber, GL_UNSIGNED_INT, 0);
}
```

Capítulo 4

Extras

4.1 Extra *Engine Features*

4.1.1 Câmera

Foi notado que para a implementação da câmera existem alguns aspetos que se têm de ter em consideração, não só como a sua posição no referencial, como também um ponto para o qual esta irá estar orientada. De maneira a garantir e simplificar o movimento de rotação da câmera opta-se pelo uso de coordenadas polares da mesma e do ponto de referência. De seguida basta converter estas novas coordenadas polares em coordenadas do referencial.

Fixed Point Rotation

Tal como é indicado, este método consiste em fixar o ponto para o qual a câmera está orientada, fazendo com que esta efetue a rotação em torno desse ponto. Para isso basta recalcular as novas coordenadas da câmera mantendo a distância desta ao centro.

Sendo assim, apenas é necessário adicionar ao ângulo α, a respetiva constante, resultante do *input* do *mouse*, para obter as novas coordenadas da câmera, as quais se encontram contidas num plano paralelo ao plano XZ, ou adicionar ao ângulo β, obtendo assim as coordenadas da câmera, relativas a um plano perpendicular ao plano XZ.

Fixed Camera Rotation

Seguindo a mesma lógica do método referido anteriormente, neste caso, é fixada a câmera e o ponto efetua a rotação em torno da câmera mantendo a mesma distância entre estes dois.

Logo, basta adicionar ao ângulo α, do ponto, a constante, tendo em conta o *input* do *mouse*, para obter as novas coordenadas do ponto, resultantes da rotação seguindo a mesma orientação que um plano paralelo ao plano XZ, ou adicionar ao ângulo β, obtendo assim as coordenadas da câmera, relativas à rotação com a mesma orientação que um plano perpendicular ao plano XZ.

Zoom

Com o intuito de efetuar o *Zoom* é necessário variar-se a distância entre o ponto, que permanece fixo, e a câmera. Para tal efeito, varia-se os valores da terceira componente das coordenadas polares da câmera, pois, mudando o raio muda-se a distância da câmera ao ponto.

Definidos os tipos de rotação da câmera é possível agora definir um método que efetue os cálculos referidos anteriormente, tendo em conta alguns limites superiores e inferiores de α e β:

```
void controlCamera(int x, int y){
        int xMove = (clickX - x);
        int yMove = (clickY - y);
        switch (moveState){
```

```
                case FCR:
                this->centerVec[0] += xMove*sensi;
                this->centerVec[1] -= yMove*sensi;
                if (abs(centerVec[1] + yMove*sensi) > (M_PI/2)){
                        this->centerVec[1] += yMove*sensi;
                }
                calcCenterP();
                break;

                case ZOM:
                if (posVec[2] + xMove*sensi > 0){
                        this->posVec[2] += xMove*sensi;
                }
                calcPosP();
                break;

                case FPR:
                this->posVec[0] += xMove*sensi;
                if (abs(posVec[1] + yMove*sensi) < (M_PI/2)){
                        this->posVec[1] += yMove*sensi;
                }
                calcPosP();
                break;

                default:
                break;
        }
        clickX = x;
        clickY = y;
        glutPostRedisplay();
}
```

Translações da câmera

Relativamente ao reposicionamento, em todas as direções e sentidos da câmera, no espaço, efetuam-se translações, ambas aplicadas à câmera como ao ponto de referência da câmera. Para tal, são atualizados os valores das coordenadas cartesianas, somando os valores atuais com os calculados de acordo com a sensibilidade proposta.

De maneira a determinar estes valores, para efetuar as translações orientadas ao longo de um plano paralelo ao do XZ, usa-se um vetor diferente do resultante da câmera e do centro, que também é paralelo ao plano XZ.

Este vetor contém a coordenada Y sempre nula, e calcula o valor das outras normalizando as coordenadas do vetor com o sentido e direção da câmera ao centro. Para efetuar a translação orientada pelo eixo Y basta usar o vetor definido por $(0, 1, 0)$.

```
void detectKeyboard(unsigned char key, int x, int y){
        calcDir();
        glutGetModifiers();
        switch (key){

                case 'w':
                        this->pos[0] += this->dirVec[0]*sensc;
                        this->pos[2] += this->dirVec[2]*sensc;
                        this->center[0] += this->dirVec[0]*sensc;
                        this->center[2] += this->dirVec[2]*sensc;
                        break;

                case 's':
                        this->pos[0] -= this->dirVec[0]*sensc;
```

```
                this->pos[2] -= this->dirVec[2]*sensc;
                this->center[0] -= this->dirVec[0]*sensc;
                this->center[2] -= this->dirVec[2]*sensc;
                break;

            case 'a':
                this->pos[0] += this->dirVec[2]*sensc;
                this->pos[2] -= this->dirVec[0]*sensc;
                this->center[0] += this->dirVec[2]*sensc;
                this->center[2] -= this->dirVec[0]*sensc;
                break;

            case 'd':
                this->pos[0] -= this->dirVec[2]*sensc;
                this->pos[2] += this->dirVec[0]*sensc;
                this->center[0] -= this->dirVec[2]*sensc;
                this->center[2] += this->dirVec[0]*sensc;
                break;

            case ' ':
                this->pos[1] += 1*sensc;
                this->center[1] += 1*sensc;
                break;

            case 'c':
                this->pos[1] -= 1*sensc;
                this->center[1] -= 1*sensc;
                break;

            case 'r':
                reset();
                break;

            default:
                break;
        }
        glutPostRedisplay();
}
```

4.1.2 *Improvements* de utilização

Nesta secção abordaremos algumas melhorias pequenas na *Engine* desenvolvidas para facilitar a locomoção do utilizador na *Scene* bem como display the alguma informação relevante para melhor experiência.

Método de desenho de Polígonos

Com "Método de desenho", é referida a maneira de se representar polígonos, seja como sólidos ou como o sistema de *Wireframe* para se verificar a malha e integridade do sólido na *Engine*.

O *Render* de polígonos é inicializado com a constante do *OpenGL* "*GL_FILL*", ou seja, ao iniciar a *Engine*, os sólidos apresentam todos polígonos preenchidos. Entretanto, a qualquer momento, através do input da tecla "m"o método passa a constante "*GL_LINE*"passando os sólidos, então, a serem representados com suas devidas *Wireframes*.

Eixos de orientação

Com todos os movimentos de câmera apresentados, para uma utilização mais consistente e menos "perdida", foi implementado um sistema de *toggle* de eixos, em duas posições críticas.

Através da tecla "o", em *runtime*, será desenhado um sistema de eixos sobre a origem, o sistema pode ser aumentado ou removido através do pressionar da mesma tecla.

A tecla "i", entretanto, reserva-se para desenhar também um sistema de eixos, mas dessa vez centrado no ponto central da câmera, isto é, o ponto para onde a câmera está olhando.

Identificação de ficheiro

Este tópico trata-se de uma facilidade para o utilizador reconhecer qual ficheiro *XML* abriu para a execução da *Engine*.

O nome do ficheiro *XML* aberto é colocado no nome da janela aberta da engine, sendo o nome então: "Engine Scene - <nameFile>.xml".

4.2 *Shape Gen* extras

Quanto Ao *Shape Generator*, algum trabalho extra desenvolveu-se sobre novos sólidos a serem criados ou novas opções de personalização aos já presentes.

4.2.1 *Cylinder*

O *Shape* do cilíndro, apesar de não requerido foi feito no gerador, utilizando-se da técnica de planificação já apresentada em conjunto com as coordenadas cilíndricas.

Cada vértice do cilíndro tem sua posição definida por uma relação dada entre o número da linha e da coluna na matriz de planificação com sua correspondência dentre as slices e stacks.

```
void shape(){
        int vtxPts = 1;
        float heiStck = height/(stacks);
        float freq = (2*M_PI)/slices;

        std::vector<int> row;
        std::vector<float> aux;
        for (int st = 0; st <= stacks+2; st++){
                row.clear();
                for (int sl = 0; sl < slices; sl++){
                        aux.clear();
                        if ( st > 0 && st < stacks+2) {
                                aux.push_back(radius*sin(freq*sl));
                                aux.push_back((-(height))/2 +(heiStck)*(st-1));
                                aux.push_back(radius*cos(freq*sl));

                                (this->vertex).push_back(aux);
                                vtxPts++;
                        }

                        else if (st == 0 && sl == slices-1){
                                aux.push_back(0);
                                aux.push_back((-(height))/2);
                                aux.push_back(0);
                                (this->vertex).push_back(aux);
                        }

                        else if (st == stacks+2 && sl == 0){
                                aux.push_back(0);
                                aux.push_back((height)/2);
                                aux.push_back(0);
                                vtxPts++;
```

```
                    (this->vertex).push_back(aux);
                }
                row.push_back(vtxPts);
            }
            (this->planar).push_back(row);
        }
}
```

4.2.2 Parâmetros adicionais

Em termos de parâmetros, foram adicionados argumentos para criação de *Shape*s mais dinâmicos do que os referidos na especificação do problema. Assim, para cada *Shape* as seguintes opções são disponibilizadas pelo gerador na respetiva ordem:

1. *Plane* - Dimensão em X, Dimensão em Z, Subdivisões em X, Subdivisões em Z;
2. *Box* - Dimensão em X, Dimensão em Y, Dimensão em Z, Subdivisões em X, Subdivisões em Y, Subdivisões em Z;
3. *Sphere* - Raio, *Slices*, *Stacks*;
4. *Cone* - Altura, Raio, *Slices*, *Stacks*;
5. *Cylinder* - Altura, Raio, *Slices*, *Stacks*;

4.3 Testes realizados e Resultados

Mostram-se a seguir alguns testes feitos (valores introduzidos) e os respectivos resultados obtidos:

```
C:\Users\luism\Desktop\Projeto CG\CG-2021\engine\build>generator plane 5 10 5 10 "plane.3d"
```

Figura 4.1: Comando que gera um plano com comprimento 10 e largura 5, com 5 divisões no eixo X e 10 divisões no eixo Z

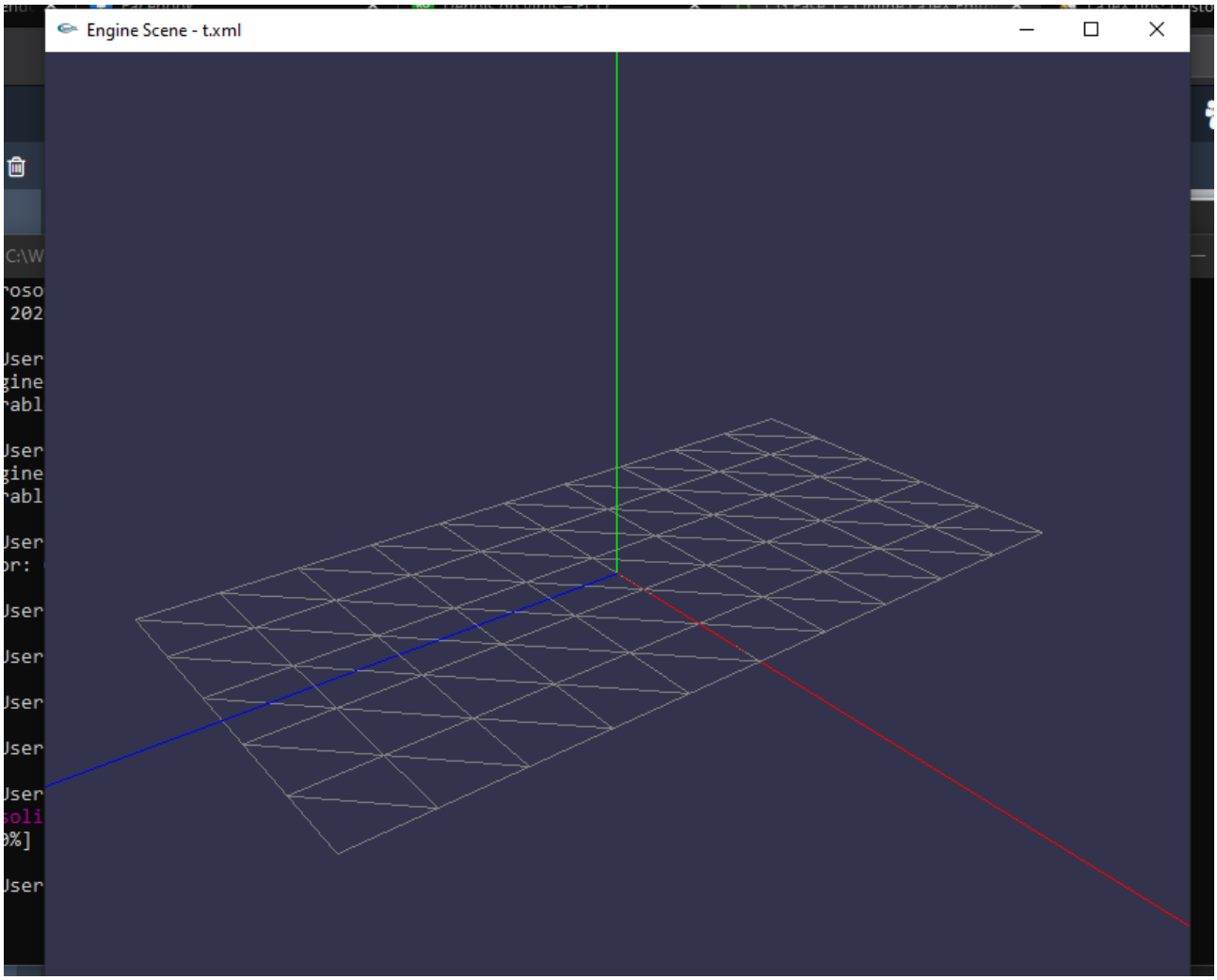

Figura 4.2: Resultado

```
:\Users\luism\Desktop\Projeto CG\CG-2021\generator\build>generator box 10 10 10 20 20 20 "box.3d"
```

Figura 4.3: Comando que gera um cubo com dimensões 10, 10, 10 (X,Y,Z), com 20 divisões por aresta

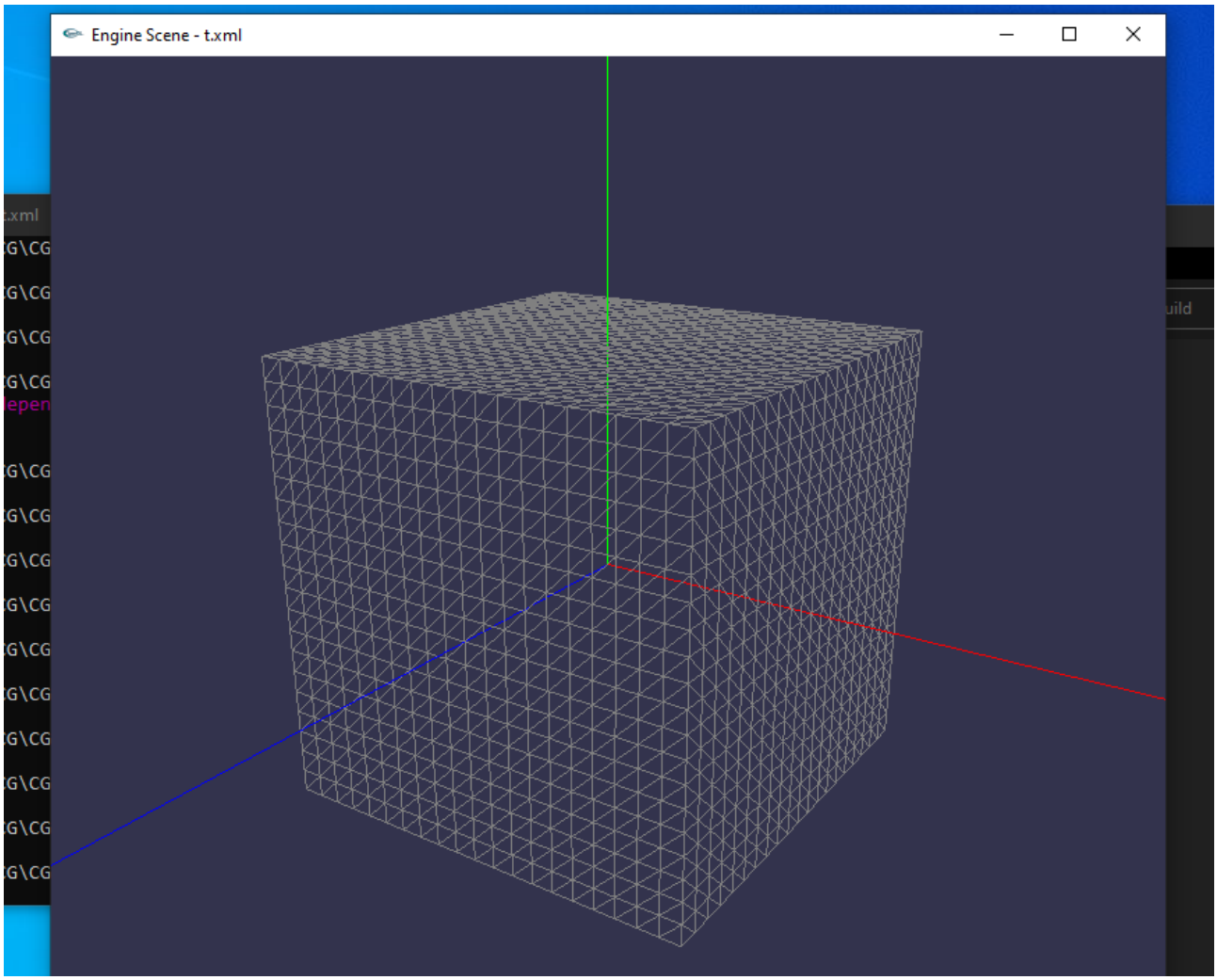

Figura 4.4: Resultado

```
C:\Users\luism\Desktop\Projeto CG\CG-2021\generator\build>generator sphere 10 50 50 "sphere.3d"
```

Figura 4.5: Comando que gera uma esfera com 10 de raio, 50 slices e 50 stacks

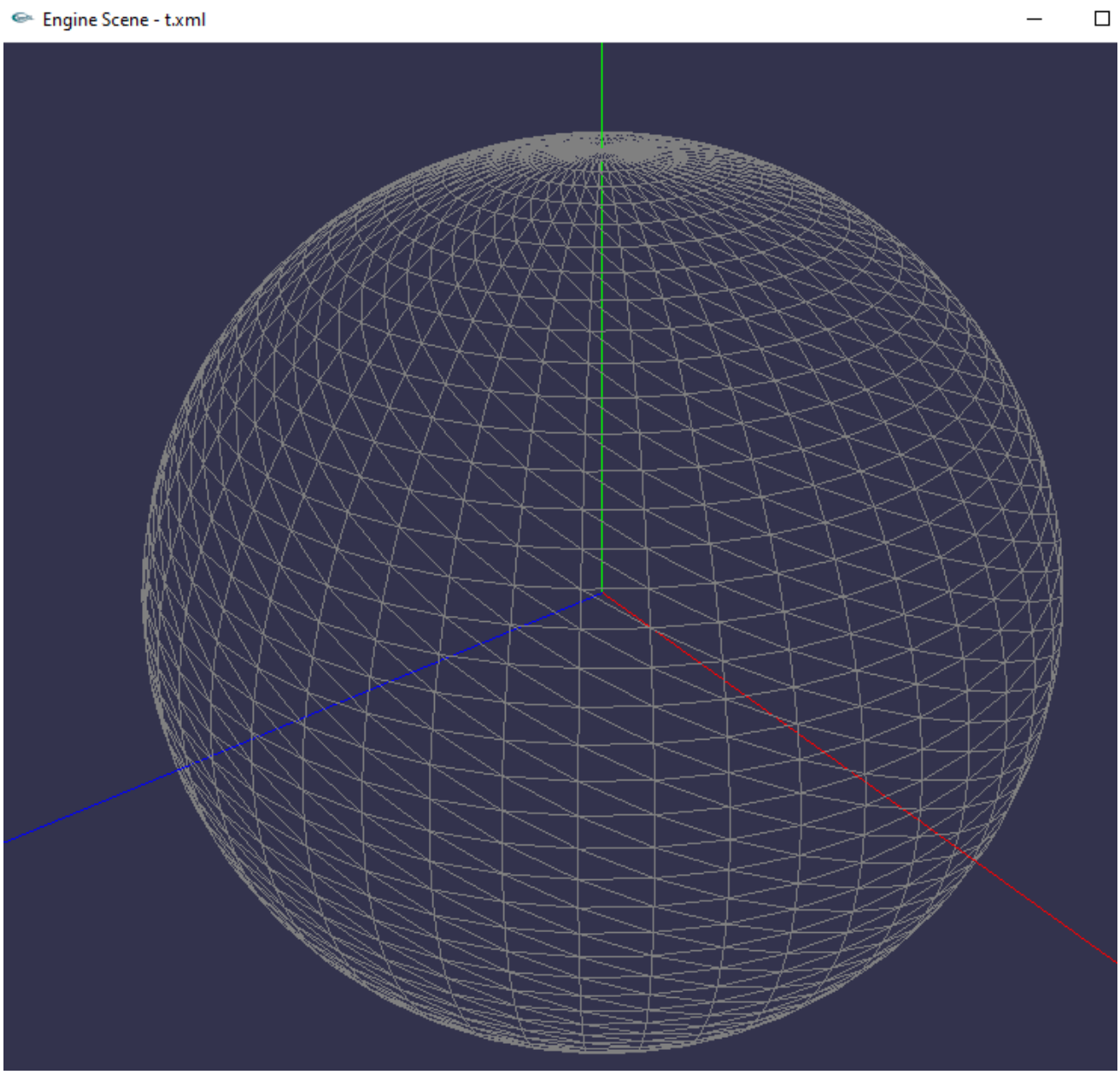

Figura 4.6: Resultado

```
C:\Users\luism\Desktop\Projeto CG\CG-2021\generator\build>generator cone 10 5 50 50 "cone.3d"
```

Figura 4.7: Comando que gera um cone com 5 de raio da base, 10 de altura, 50 slices e 50 stacks

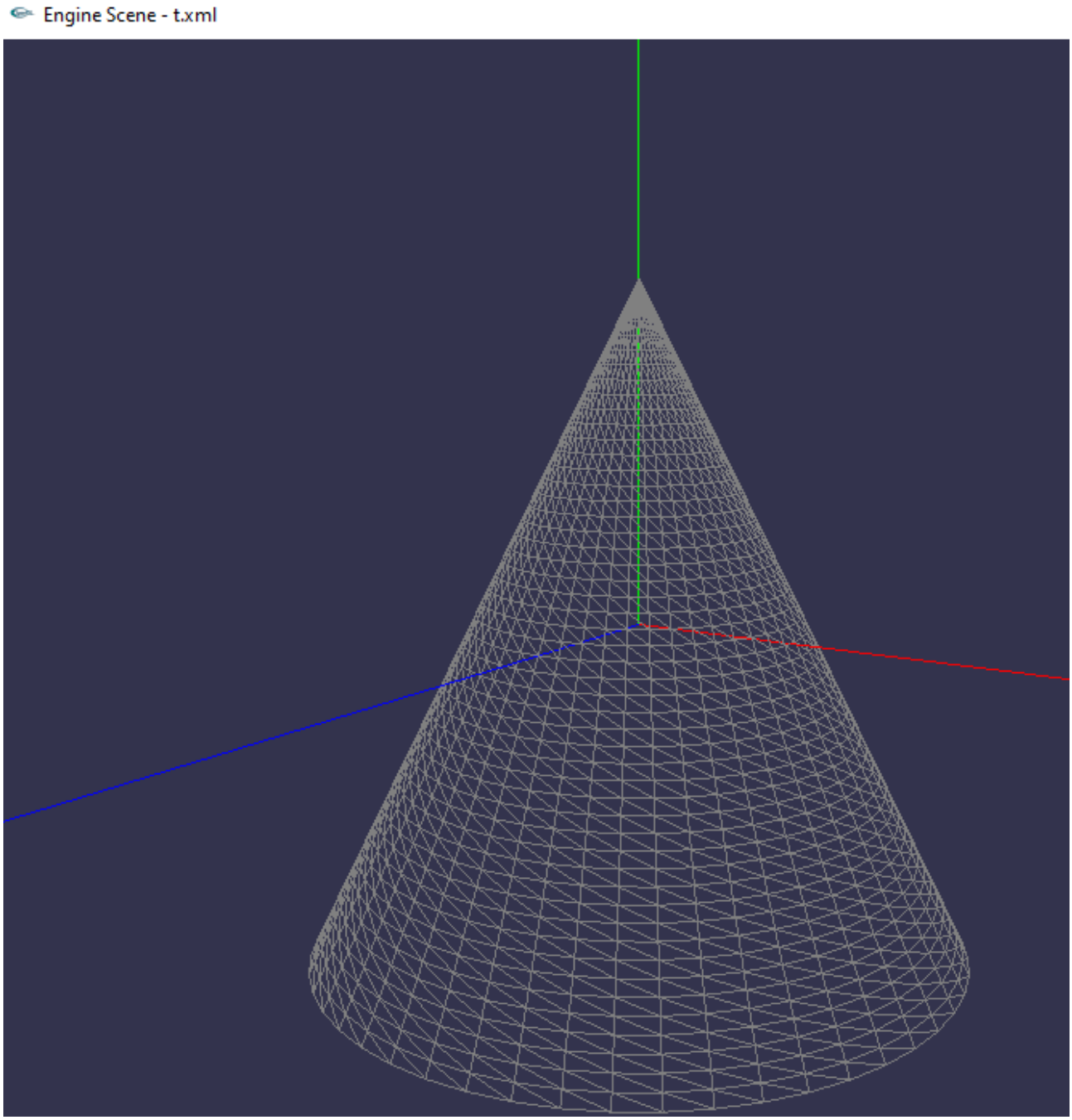

Figura 4.8: Resultado

```
C:\Users\luism\Desktop\Projeto CG\CG-2021\generator\build>generator cylinder 10 5 50 50 "cyl.3d"
```

Figura 4.9: Comando que gera um cilíndro com 10 de altura, 5 de raio, 50 slices e 50 stacks

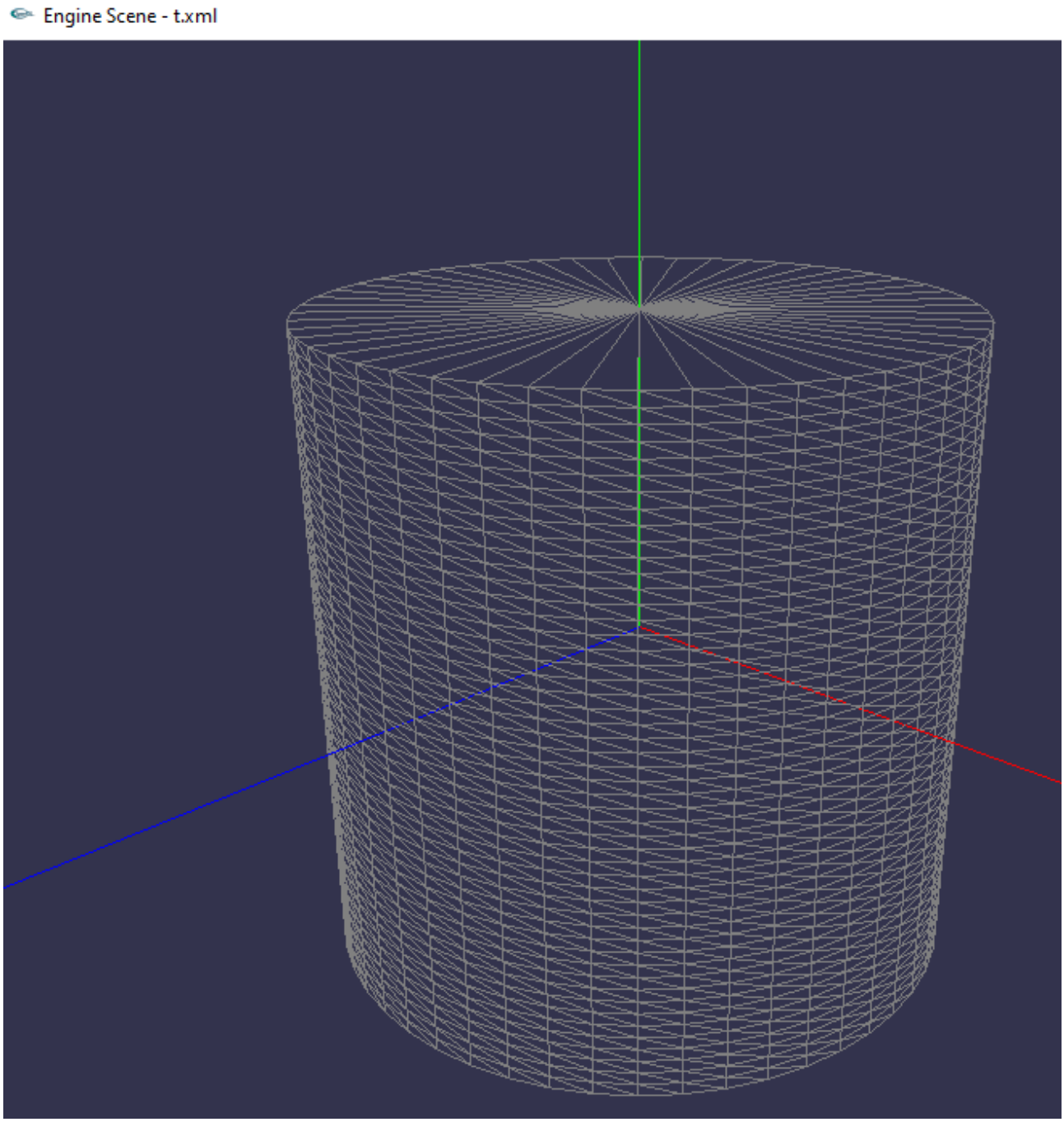

Figura 4.10: Resultado

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

Figura 4.11: Ficheiro XML que junta todas as primitivas geradas

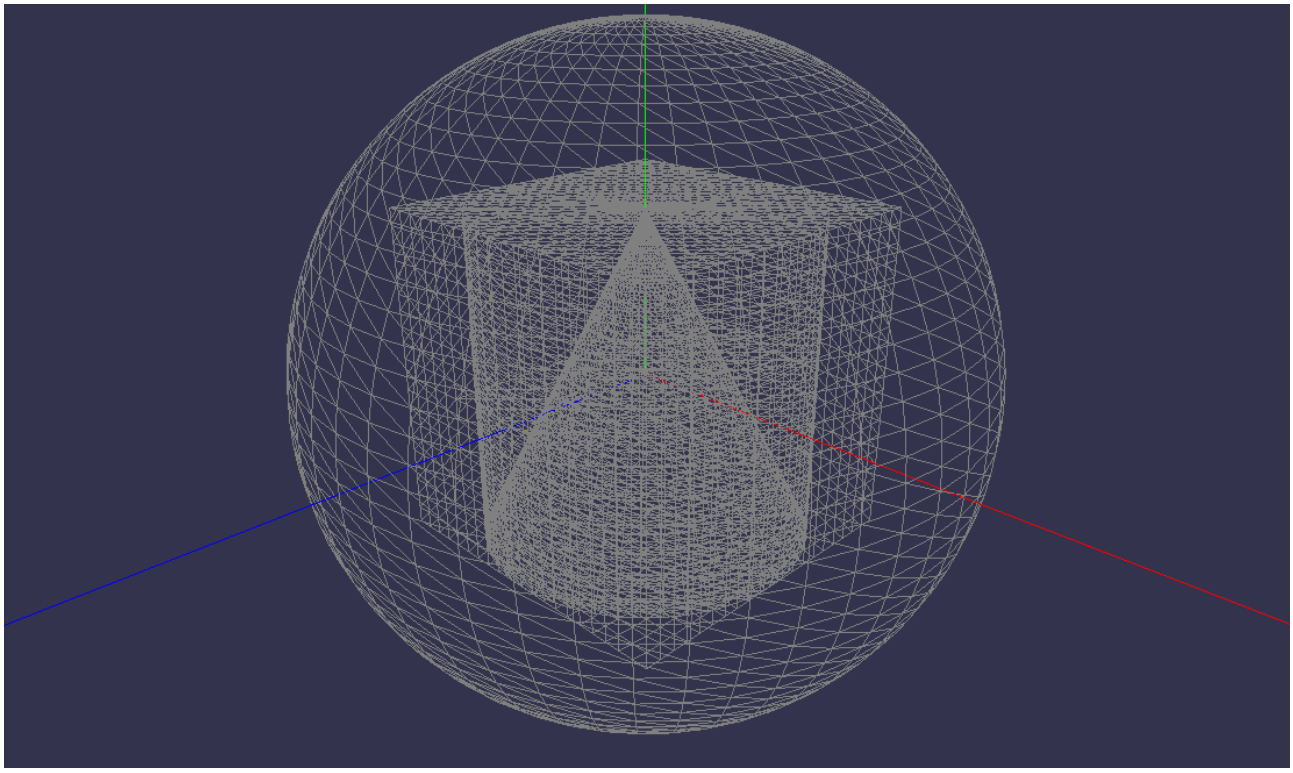

Figura 4.12: Resultado

Capítulo 5

Conclusão

Observando os programas desenvolvidos (*engine* e *generator*) assim como as soluções adotadas para resolver os vários problemas que surgiram, acreditamos que foram implementadas as soluções mais eficientes em que pensamos.

Poderiam ter sido utilizadas outras soluções, por exemplo para a geração do cubo, pois acabamos por gerar 6 planos e apenas rodar cada um deles para obtermos as faces do mesmo. Este método introduz repetição de pontos, o que não é o ideal.

Visto que estamos a utilizar *VBOs*, poderíamos também libertar a memória correspondente à lista de vértices e à lista de faces de cada objeto *Model* após carrega-las para VRAM. No entanto não o fizemos pois esta informação pode ser necessária em RAM para as futuras fases do trabalho prático.