Universidade do Minho

Mestrado Integrado em Engenharia Informática

Computação Gráfica

Fase 1

Grupo 32

César Henriques - A64321

João Gomes - A74033

Rafael Magalhães - A75377

Tiago Fraga - A74092

6 de Março de 2017

Conteúdo

1 Introdução **2**

2 Gerador **3**
2.1 Variáveis e Estruturas 3
2.1.1 Estrutura para guardar coordenadas 3
2.1.2 Matriz *quadrantes* 3
2.2 As funções que fazem o trabalho todo 4
2.2.1 Função que gera o Plano 4
2.3 Box 5
2.4 Sphere 5
2.5 Cone 7

3 Motor 3D **8**
3.1 Estruturas 8
3.2 Ficheiros 8
3.2.1 XML 8
3.2.2 Ficheiro com Vértices 9
3.3 Leitura dos Modelos 9
3.4 Desenhar 10

4 Demos das Primitivas **11**
4.1 Plane 11
4.2 Box 12
4.3 Sphere 13
4.4 Cone 14

5 Conclusão **17**

1 Introdução

Aqui abordaremos os aspetos mais importantes da primeira fase do nosso trabalho, onde fazemos uma síntese e explicação do código elaborado e dos outputs gerados. Este relatório esta dividido em duas importantes partes.

Em primeiro lugar, descrevemos a elaboração do código, com uma análise pormenorizada sobre como funciona a aplicação do gerador de vertices, apresentando diagramas para facilitar na compreensão.

Em segundo lugar, explicamos como são carregados os modelos para o nosso motor e apresentamos algumas imagens com os mesmos.

2 Gerador

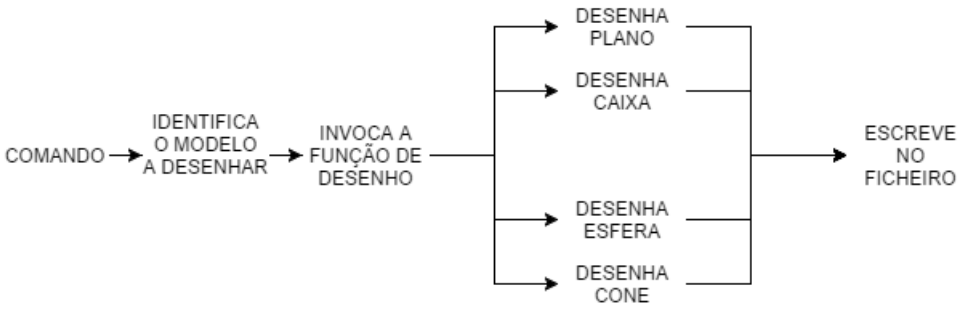


Figura 1: Diagrama do Funcionamento da Aplicação Generator

2.1 Variáveis e Estruturas

2.1.1 Estrutura para guardar coordenadas

Uma struct que contém um array de floats com 3 posições que correspondem as coordenadas X,Y e Z.

2.1.2 Matriz *quadrantes*

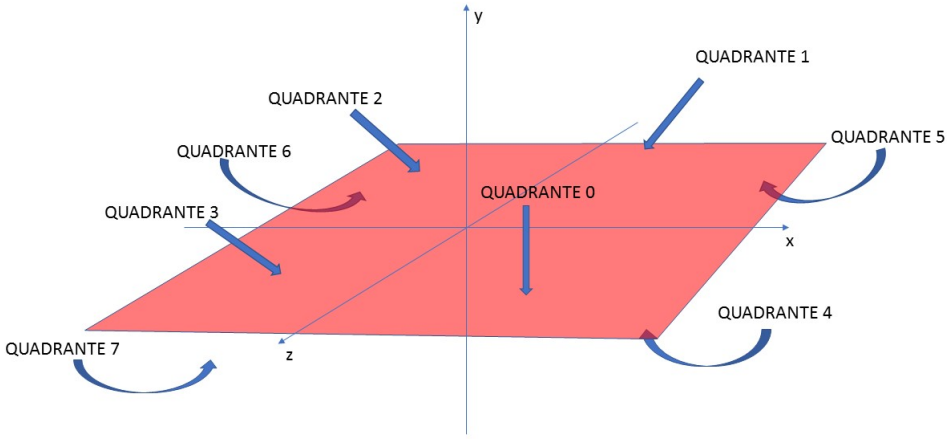
Pensamos nesta matriz como um array de vetores unitários. Cada vetor dá o sentido para o qual queremos desenhar o vertice.

Coordenadas / Quadrantes	X	Y	Z
0	1	1	1
1	1	1	-1
2	-1	1	-1
3	-1	1	1
4	1	-1	1
5	1	-1	-1
6	-1	-1	-1
7	-1	-1	1

Figura 2: Valores da matriz Quadrantes

Os quadrantes estão numerados da seguinta forma:

Figura 3: Numeração dos Quadrantes



2.2 As funções que fazem o trabalho todo

A ideia destas funções é preencher um vetor com pontos. Este é depois passado como argumento para a função que escreve esses pontos para o ficheiro. A ordem pela qual vamos gerando os vértices vai depender de qual face desejamos mostrar (*Clockwise ou Counter-Clockwise*).

2.2.1 Função que gera o Plano

```
int generateModelPlane(float size, char* fileName);
```

Para desenhar o plano, fazemos uso da matriz de quadrantes que criamos. Para isso, declaramos um array de inteiros onde vamos colocar a ordem dos quadrantes onde queremos desenhar um ponto.

$$intsequencia\_quadrantes[6] = \{0, 1, 3, 3, 1, 2\}; \tag{1}$$

Então, vamos desenhar primeiro um ponto no quadrante 0, depois no quadrante 1, etc...

A complementar a estrutura dos quadrantes, temos as três equações:

$$x = \frac{tamanho}{2} * coordenadaXdoquadrante[i]; \tag{2}$$

$$y = 0; \tag{3}$$

$$z = \frac{tamanho}{2} * coordenadaZdoquadrante[i]; \tag{4}$$

i varia a cada iteração de forma a percorrer o array com a sequência de quadrantes.

Por fim, é invocada a função que escreve todos os pontos no ficheiro.

2.3 Box

```
int generateModelBox (float xdim, float ydim, float zdim, char* fileName);
```

Assim como na função acima utilizamos uma estrutura auxiliar para saber a ordem dos quadrantes onde desenhar.
Para a construção da caixa baseamos-nos nas seguintes equações algébricas:

$$x = \frac{xdim}{2} * coordenadaXdoquadrante[i]; \tag{5}$$

$$y = \frac{ydim}{2} * coordenadaYdoquadrante[i]; \tag{6}$$

$$z = \frac{zdim}{2} * coordenadaZdoquadrante[i]; \tag{7}$$

Basicamente, estamos apenas a desenhar planos, logo as equações a desenhar são idênticas.

2.4 Sphere

```
int generateModelSphere (float radius, unsigned int slices,
unsigned int stacks, char* fileName);
```

Para desenhar a esfera, temos um ciclo exterior para cada camada e um ciclo interior para cada fatia. A cada iteração do ciclo interior desenhamos seis vértices (dois triângulos). Calculamos os pontos a partir das coordenadas polares de uma esfera.

$$x = raio * cos(beta) * sin(alpha); \tag{8}$$

$$y = raio * sin(beta); \tag{9}$$

$$z = raio * cos(beta) * cos(alpha); \tag{10}$$

É também importante salientar que, para desenhar o triângulos, a cada iteração é calculado o próximo alpha e o próximo beta.

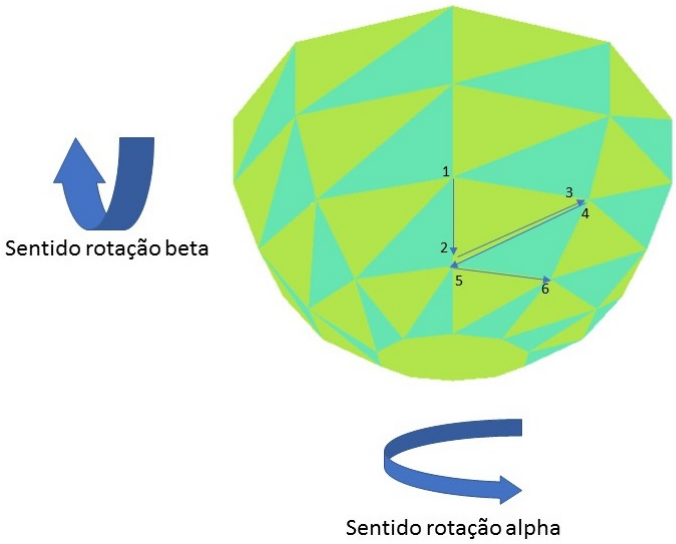


Figura 4: Ordem de desenho dos pontos

Para facilitar os cálculos, e como dois vértices são sempre comuns aos dois triângulos a gerar por iteração, criamos duas variáveis:

```
Point vertice_comum1, vertice_comum2;
```

O objetivo é guardar a informação relativa aos dois pontos comuns e inserir no vetor, na devia ordem. Tendo a Figura 4 como exemplo, são calculados os pontos 1, 2 e 3. Como 2 e 3 são comuns ao próximo triângulos são logo inseridos no vetor.

```
vertice.coordenadas[0] = ...
pontos.push_back(vertice);

vertice_comum1.coordenadas[0] = ...
pontos.push_back(vertice_comum1);

vertice_comum2.coordenadas[0] = ...
pontos.push_back(vertice_comum2);
pontos.push_back(vertice_comum2);
pontos.push_back(vertice_comum1);
```

Após gerar todos os pontos, é invocada a função de escrita no ficheiro.

2.5 Cone

```
int generateModelCone(float radius, float height, unsigned int slices,
unsigned int stacks, char* fileName);
```

O cone é idêntico à esfera, só que desta vez o raio vai decrementando à medida que vamos subindo nas camadas. Ao dividirmos o raio pelo numero de camadas vamos obter o valor que será subtraído ao raio por iteração. De igual forma, foi necessário também calcular o próximo alpha e beta a cada iteração.

Após desenhar a parte de cima do cone, usamos mais um ciclo para gerar a base. Esta foi desenhada no sentido contrário ao do cone (alpha negativo). Sem a base, teríamos uma figura parecida com um chapéu de aniversário.

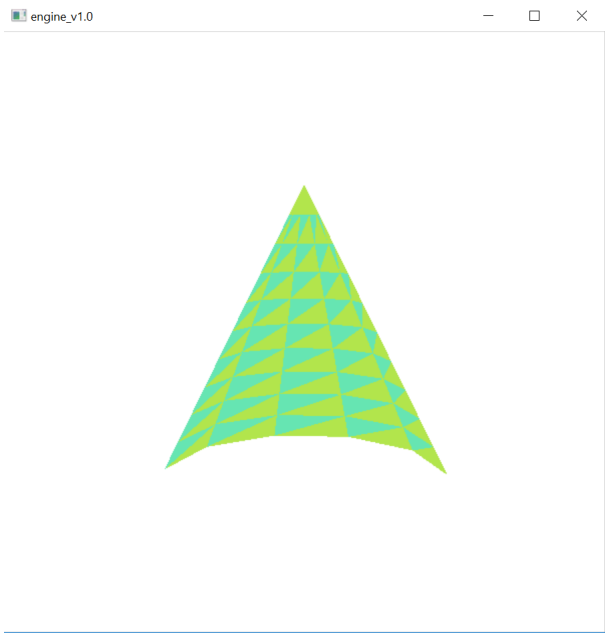


Figura 5: Cone sem base.

3 Motor 3D

3.1 Estruturas

- `Typedef vector<String> Primitive;`
- `Vector<Primitive> models;`

Definimos o tipo Primitiva como um vetor de strings, em que cada string corresponde a um vertice.

Para armazenar as primitivas recorremos a um vetor de Primitive. Este vetor vai conter no seu estado final todas as primitivas carregadas a partir dos ficheiros.

3.2 Ficheiros

3.2.1 XML

Um exemplo básico de um ficheiro possivel de XML seria:

config.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <scene>
3    <model file="cone.3d"/>
4  </scene>
5
```

Figura 6: Exemplo de ficheiro XML

A aplicação é capaz de ler mais modelos de uma só vez, no entanto, todas as primitivas são geradas na origem, logo iriam ficar sobrepostas.

3.2.2 Ficheiro com Vértices

O formato dos ficheiros dos modelos gerados pelo gerador é simples. Na primeira linha incluímos o numero de vértices gerados. As restantes linhas contêm pontos, separados por virgulas onde cada linha corresponde a um ponto.

config.xml | sphere.3d

```
1  300
2  0.000000, -1.902113, 0.618034
3  -0.000000, -2.000000, -0.000000
4  0.363271, -1.902113, 0.500000
5  0.363271, -1.902113, 0.500000
6  -0.000000, -2.000000, -0.000000
7  -0.000000, -2.000000, -0.000000
8  0.363271, -1.902113, 0.500000
```

Figura 7: Exemplo de Ficheiro com Vertices

3.3 Leitura dos Modelos

```
int loadXML();
int loadModels(vector<String> &list);
```

Para fazer o load e parsing do ficheiro XML utilizamos a biblioteca TinyXML2 disponivel na internet(*grinninglizard.com/tinyxml2/*). Todos os ficheiros são lidos e carregados apenas uma vez no inicio da aplicação.

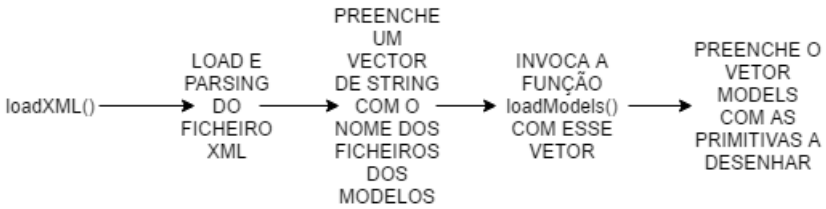


Figura 8: Processo de Armazenamento das Primitivas

3.4 Desenhar

```
Void drawPrimitive (const Primitive& primitive);
```

A drawPrimitive() é invocada, na função que desenha a cena, enquanto o vetor models tiver primitivas a desenhar. No seu coração está um for loop que divide e converte uma linha de pontos em três variáveis float x, y e z que são usados com o método *glVertex3f(x,y,z)*. O ciclo dura enquanto o numero de iterações não for igual ao número de vértices. A cor a desenhar é alterada a cada triângulo.

4 Demos das Primitivas

4.1 Plane

Um plano de lado e comprimento 2.

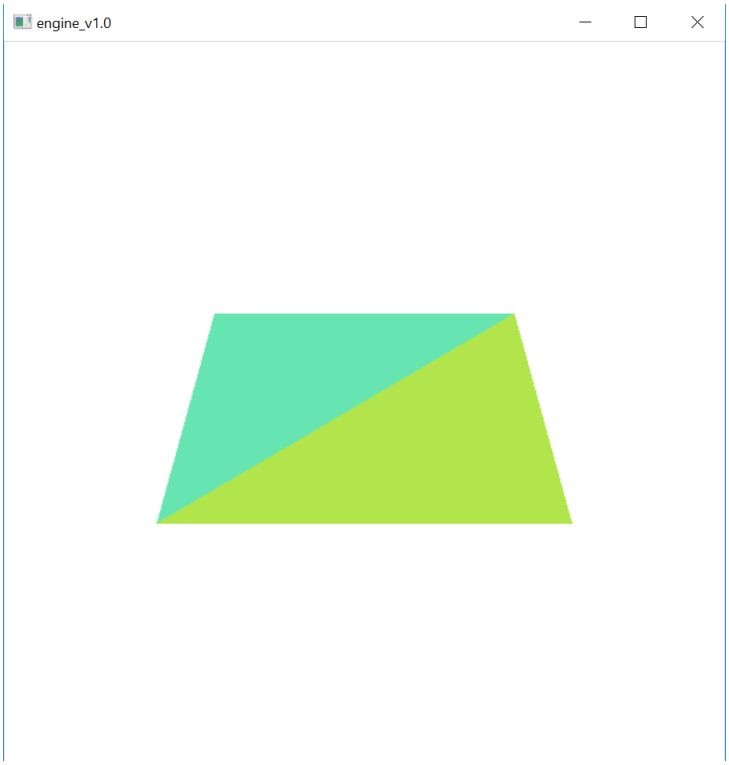


Figura 9: Plano.

4.2 Box

Uma box com comprimento 2, largura 2 e altura 2.

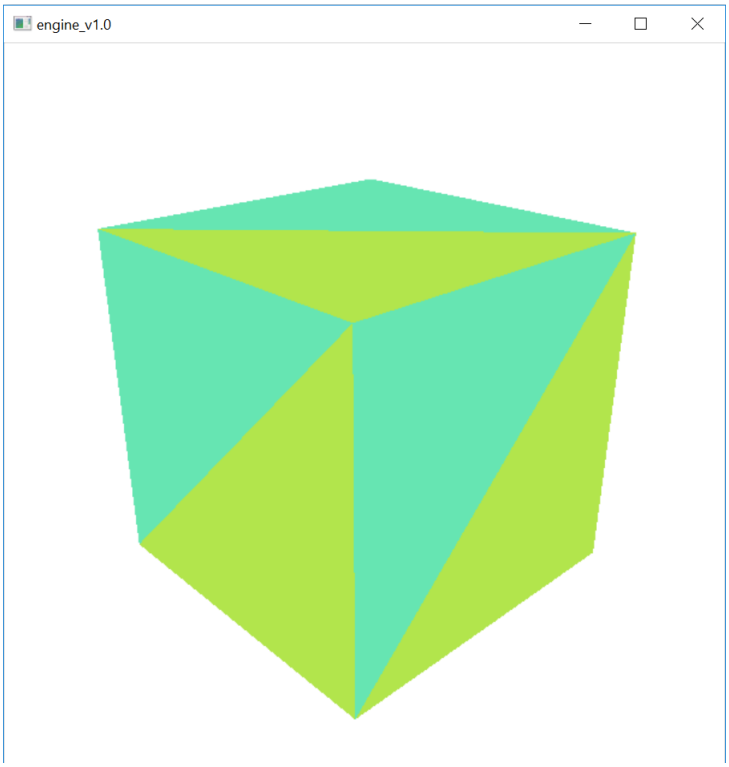


Figura 10: Box

4.3 Sphere

Aqui demonstramos uma esfera com raio 2, 10 fatias e 10 camadas.

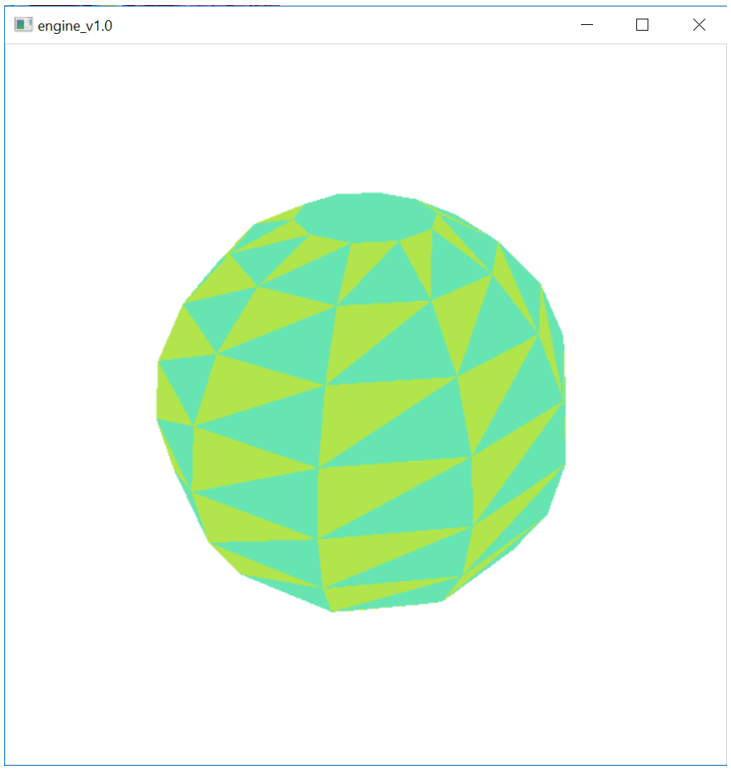


Figura 11: Esfera de raio 2, 10 stacks e 10 splices.

4.4 Cone

Um cone de raio 1, altura 2, 10 camadas e 10 fatias.

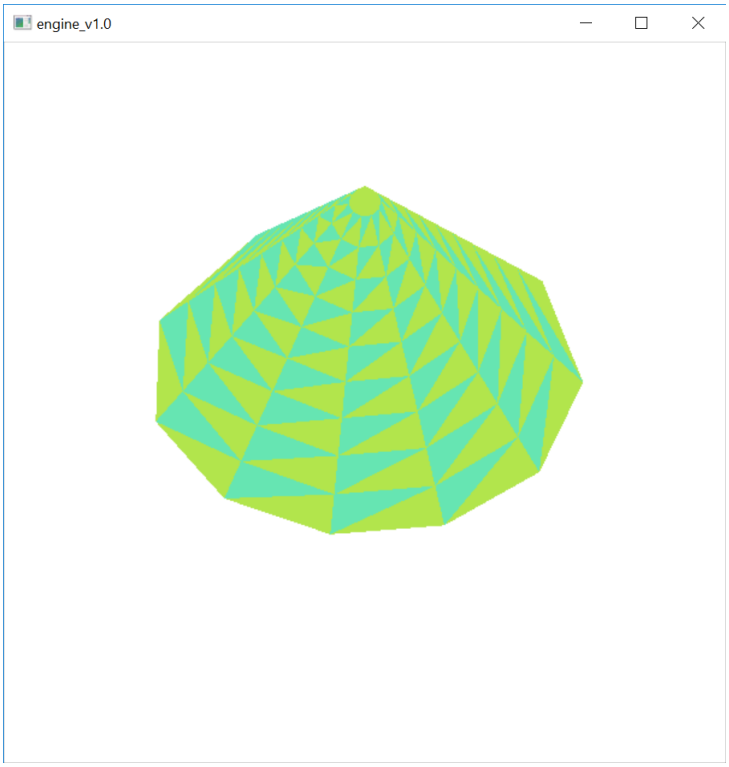


Figura 12: cone

Algumas das capacidades do nosso motor. Um cone de raio 1, altura 2, 100 fatias e 100 camadas com a opção de menu em wired.

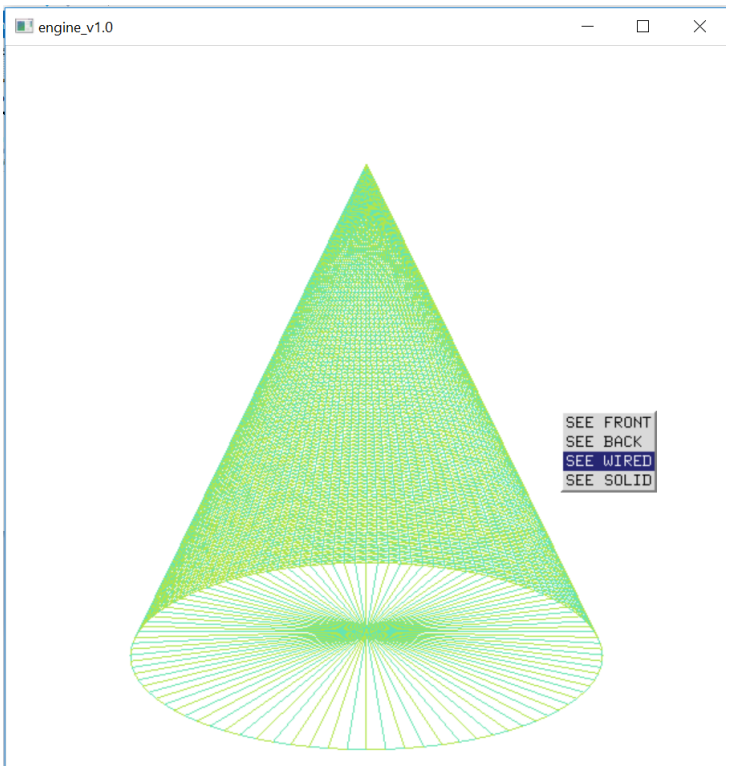


Figura 13: Cone Wire.

E uma esfera de raio 2, com 100 fatias e 100 camadas também em wire.

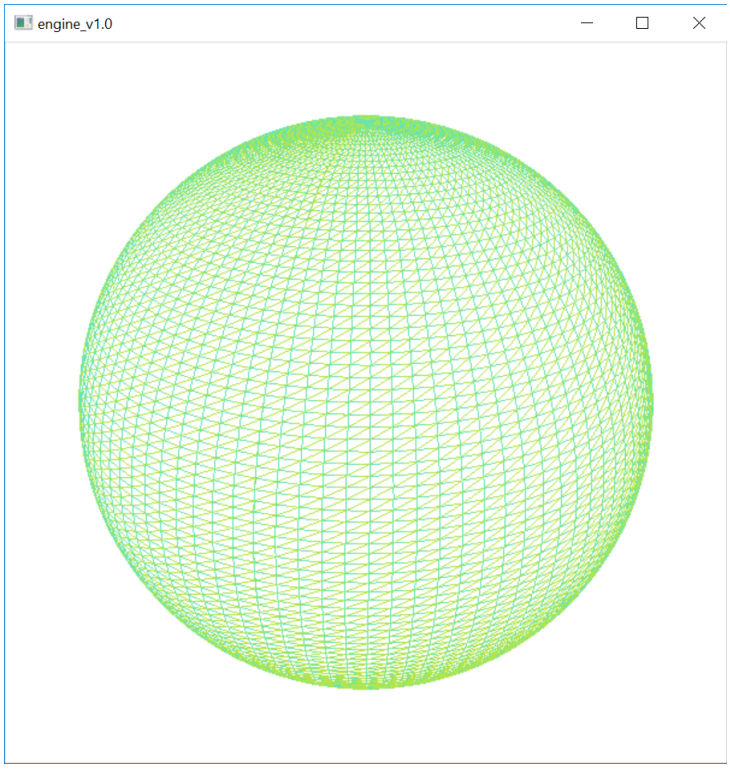


Figura 14: Esfera Wire.

5 Conclusão

No geral, conseguimos cumprir os objetivos estabelecidos para esta etapa: criar um gerador e um motor 3D para desenhar os modelos gerados. O gerador funciona para todos os modelos, ficando por implementar a capacidade de gerar a Box com diferentes subdivisões. O motor lê e interpreta um ficheiro XML, guarda a informação necessária e, após isso, desenha o modelo. Implementamos também a movimentação da camera (rotação com as setas e zoom com as teclas M e N) e um menu com algumas funcionalidades básicas. Podiamos, por exemplo, ter também implentado um mostrador de FPS. Terminamos assim a primeira fase do projeto.