



Universidade do Minho
Escola de Engenharia

Trabalho Prático - Primitivas Gráficas - Fase 1 Grupo 32

Computação Gráfica

Bruno Filipe de Sousa Dias A89583

Luís Enes Sousa A89597

Pedro Miguel de Soveral Pacheco Barbosa A89529



14 de março de 2021

Conteúdo

1	Introdução	1
2	Estrutura do Projeto	2
2.1	Aplicações	2
2.1.1	Generator	2
2.1.2	Engine	2
2.2	Classes	2
2.2.1	Ponto	2
2.3	Ferramentas Auxiliares	3
2.3.1	TinyXML2	3
3	Primitivas Geométricas	4
3.1	Plano	4
3.2	Box	5
3.3	Esfera	7
3.4	Cone	9
4	Generator	12
4.1	Descrição	12
4.2	Funcionalidades	12
4.3	Demonstração	12
5	Engine	14
5.1	Descrição	14
5.2	Funcionalidades	14
5.3	Demonstração	15
6	Modelos 3D	16
6.1	Plano	16
6.2	Box	18
6.3	Esfera	20
6.4	Cone	22
7	Conclusão e Trabalho Futuro	24

1 Introdução

No âmbito da Unidade Curricular de Computação Gráfica, foi-nos proposto o desenvolvimento de um motor de representação gráfico 3D e fornecer exemplos de uso que demonstrassem o seu potencial.

O trabalho prático foi dividido em quatro fases de entrega, sendo que a primeira consiste em montar a arquitetura que vai servir como base para todo o nosso sistema. Deste modo, foram criadas duas aplicações: uma para gerar os ficheiros com as informações dos modelos, à qual chamamos de *Generator*; e outra que lê os ficheiros XML e representa um cenário recorrendo aos ficheiros gerados, à qual chamamos de *Engine*.

Nesta primeira fase, foi apenas pedida a correta interpretação e representação de primitivas gráficas básicas: o plano, a box, a esfera e o cone.

O projeto é desenvolvido na linguagem C++ e recorrendo à biblioteca OpenGL.

2 Estrutura do Projeto

Nesta secção do relatório iremos abordar a forma como o nosso projeto está estruturado, com base nas Aplicações, Classes e Primitivas Gráficas que desenvolvemos nesta primeira fase do trabalho prático.

2.1 Aplicações

Nesta secção serão apresentadas as aplicações fundamentais que permitem gerar e representar os diferentes modelos.

2.1.1 Generator

O módulo **Generator.cpp** é responsável por gerar todas as primitivas gráficas e recebe como argumentos o tipo da primitiva gráfica a ser criada bem como outros parâmetros necessários à sua criação. Desta forma, calcula todos os pontos necessários para formar os diferentes triângulos que irão formar a figura pretendida e grava-os no ficheiro desejado.

O módulo **Primitivas.cpp** é responsável por conter todos os algoritmos necessários para a criação dos vértices necessários para gerar as diversas formas geométricas pedidas nesta fase do projeto.

2.1.2 Engine

O módulo **Engine.cpp** é uma aplicação que permite a interpretação e a representação do ficheiro *XML* que contém o nome dos ficheiros gerados pelo **Generator** numa modelação 3D. O ficheiro *XML* deverá conter todos os ficheiros com a extensão *.3D* necessários para a construção das diferentes primitivas geométricas.

2.2 Classes

De maneira a facilitar o desenvolvimento das aplicações acima referidas, decidimos criar algumas classes que servem de suporte para esse efeito.

2.2.1 Ponto

A classe Ponto está definida em **Ponto.cpp** e serve para facilitar a passagem da representação de um ponto num referencial para código. Desta forma, cada ponto vai ser representado pelas coordenadas x, y e z que são três floats.

2.3 Ferramentas Auxiliares

Para facilitar a realização do trabalho prático recorreremos à utilização de algumas ferramentas auxiliares.

2.3.1 TinyXML2

Como auxílio para o nosso trabalho, e para além da biblioteca *OpenGL* utilizada para as funcionalidades gráficas deste projeto, utilizamos também a biblioteca ***TinyXML2*** que facilitou o *parsing* dos ficheiros XML.

3 Primitivas Geométricas

3.1 Plano

O **Plano** irá ser formado por dois triângulos diferentes que no seu conjunto irão formar um quadrado. Por sua vez, estes dois triângulos irão partilhar dois dos seus pontos, os que formam a hipotenusa dos mesmos.

Como pretendemos desenhar o plano contido no plano XZ então todos os vértices criados irão ter a sua coordenada Y a 0. É de realçar também a necessidade de se ter em atenção a ordem com que os vértices são criados de forma ao plano ser visto de cima. Para isto é necessário recorrer à técnica da mão direita (o polegar fica direcionado para cima e fechando a mão percebemos em que sentido os vértices devem ser criados a partir do momento em que criamos um primeiro vértice qualquer).

Tendo agora pensado nos dois triângulos que irão formar um quadrado e tendo em consideração que o *size* corresponde ao tamanho de um lado desse quadrado, criamos então uma variável *distância* que será igual ao *size*/2 de modo a que o nosso plano fique centrado na origem.

Assim, os vértices dos triângulos gerados terão as seguintes coordenadas:

$$distância = size/2;$$

- **Triângulo 1**

- Ponto 1 : (*distância*, 0, *distância*)
- Ponto 2 : (*distância*, 0, -*distância*)
- Ponto 3 : (-*distância*, 0, -*distância*)

- **Triângulo 2**

- Ponto 1 : (-*distância*, 0, -*distância*)
- Ponto 2 : (-*distância*, 0, *distância*)
- Ponto 3 : (*distância*, 0, *distância*)

3.2 Box

Para formar uma **Box** temos de perceber os aspetos principais que o compõe. Uma box tem um comportamento completamente análogo ao de um cubo, no entanto, pode ter dimensões diferentes nas diferentes faces e não ser propriamente formado por quadrados em todas elas.

Deste modo, sabemos então que uma box é constituído por 6 faces (compostas tanto por quadrados como por retângulos). Dessa maneira iríamos precisar de ter pelo menos 2 triângulos em cada face para ser possível desenhar a totalidade do cubo. Existe no entanto a possibilidade de o utilizador decidir colocar um dado número de divisões por aresta á sua escolha. Deste modo olhando para uma face de um cubo, caso esta fosse um quadrado, consoante o numero de divisões pedidas iria desmonstar-se em quadrados mais pequenos (se fossem 2 divisões iriam ser 4 quadrados mais pequenos, se fossem 3 divisões, 9 quadrados, etc.)

Assim a construção de pontos é formada por 3 fases, funcionando todas da mesma maneira. Cada fase é constituída por 2 ciclos do tipo *for* e diz respeito às 2 faces da box paralelas a cada um dos planos XZ , XY e YZ .

Explicando para as faces paralelas ao plano XZ :

- Cada *for* vai ser percorrido o número de divisões pedidas para cada aresta. Em cada iteração do *for* interior vai ser desenhado um quadrado de cada vez de cada lado do plano em estudo (são então desenhados 4 triângulos de cada vez). Em cada iteração do *for* exterior, dado que foram executadas n vezes o *for* interior (sendo n o número de divisões por aresta) vai ser construída uma tira (conjunto de quadrados/retangulos) mais uma vez, de cada lado do plano em estudo.
- Precisamos de valores iniciais para sabermos onde iremos começar cada iteração, estes valores vão ser do tipo: (inicial X = dimensão X / 2)
- $\text{Ponto}(-\text{inicial}X + \text{div}X * \text{distancia}X, \text{inicial}Y, -\text{inicial}Z + (\text{div}Z + 1) * \text{distancia}Z);$
 $\text{Ponto}(-\text{inicial}X + (\text{div}X + 1) * \text{distancia}X, \text{inicial}Y, -\text{inicial}Z + \text{div}Z * \text{distancia}Z);$
 $\text{Ponto}(-\text{inicial}X + \text{div}X * \text{distancia}X, \text{inicial}Y, -\text{inicial}Z + \text{div}Z * \text{distancia}Z);$

 $\text{Ponto}(-\text{inicial}X + \text{div}X * \text{distancia}X, \text{inicial}Y, -\text{inicial}Z + (\text{div}Z + 1) * \text{distancia}Z));$
 $\text{Ponto}(-\text{inicial}X + (\text{div}X + 1) * \text{distancia}X, \text{inicial}Y, -\text{inicial}Z + (\text{div}Z + 1) * \text{distancia}Z);$
 $\text{Ponto}(-\text{inicial}X + \text{div}X * \text{distancia}X, \text{inicial}Y, -\text{inicial}Z + \text{div}Z * \text{distancia}Z);$

- Neste caso seria formado um quadrado do lado positivo do plano XZ . Dentro do mesmo loop interior iria ser também formado um quadrado do lado negativo do plano XZ , alterando a coordenada Y dos pontos para $-inicialY$ e verificando a ordem dos pontos, para todos estes triângulos serem visíveis do lado de fora.
- A cada loop a divisão a que o loop diz respeito aumenta, ou seja, se estivermos a estudar o loop da divisão dos Z iríamos ter:

for (int divZ = 0; divZ < nrDivisoas; divZ ++)

Em conjunto com o aumento do número da divisão em que nos encontramos, é também utilizado um float denominado *distanciaZ* que representa o shift que temos de dar para formar os diferentes pontos. Esta distância/shift é encontrado com:

float distanciaZ = dimZ/nrDivisoas;

O mesmo aconteceria para as faces paralelas ao plano XY e YZ . Todo o comportamento demonstrado iria ser respeitado de igual forma. A única diferença é as variáveis utilizadas.

Assim, ficamos a perceber que a formação de um cubo passa pela construção de 6 faces. Cada face pode ser constituída unica e exclusivamente por 2 triângulos, ou então por um maior número de triângulos, caso o Utilizador deseje que existam divisões. Nesse caso, cada face terá $2 * 2^{nrDivisoas}$ triângulos em cada face.

3.3 Esfera

De forma a gerar os diferentes vértices que criam os triângulos que, na sua globalidade, formam uma **Esfera** foi necessário dividir esta em fatias ou slices (na vertical) e camadas ou stacks (na horizontal). Também foi necessário definir um raio.

Deste modo, para representar cada um dos vértices do triângulo foi necessário definir as coordenadas esféricas que são dadas pelas seguintes equações:

- **Raio** = Raio Esférico
- $\mathbf{x} = \text{raio} * \cos(\beta) * \sin(\alpha)$
- $\mathbf{y} = \text{raio} * \sin(\beta)$
- $\mathbf{z} = \text{raio} * \cos(\beta) * \cos(\alpha)$

Os diferentes pontos dos triângulos são obtidos a partir de diferentes variações dos ângulos β e α . Se o nosso objetivo for mudar a camada na qual estamos a construir os triângulos mudamos o valor de β . Se o objetivo for mudar a fatia então variamos o valor de α .

De seguida mostramos as equações utilizadas para calcular os valores dos ângulos α e β .

- $\alpha = \text{númerofatia} * (2 * \pi / \text{númerototalfatias})$
- $\beta = - (\pi / 2)$

Para obtermos o ângulo α seguinte **proxAnguloFatia** adicionamos-lhe $(2 * \pi / \text{númerototalfatias})$.

Para obtermos o ângulo β seguinte **proxAnguloStack** adicionamos-lhe $(\pi / \text{númerototalcamadas})$.

Tendo este conhecimento adquirido a estratégia utilizada para construir a esfera foi a seguinte: selecionando apenas uma camada e uma fatia formamos um quadrado formado por 2 triângulos para as camadas intermédias e a primeira e última camadas são formadas por um só triângulo.

Assim para a **camada inferior** os pontos do triângulo serão os seguintes:

- **Triângulo Camada Inferior**

- Ponto 1 : $(0, -raio, 0)$
- Ponto 2 : $(raio * \cos(\text{proxAnguloStack}) * \sin(\text{proxAnguloFatia}), raio * \sin(\text{proxAnguloStack}), raio * \cos(\text{proxAnguloStack}) * \cos(\text{proxAnguloFatia}))$
- Ponto 3 : $(raio * \cos(\text{proxAnguloStack}) * \sin(\text{anguloFatia}), raio * \sin(\text{proxAnguloStack}), raio * \cos(\text{proxAnguloStack}) * \cos(\text{anguloFatia}))$

Assim para a **camada superior** os pontos do triângulo serão os seguintes:

- **Triângulo Camada Superior**

- Ponto 1 : $(0, -raio, 0)$
- Ponto 2 : $(raio * \cos(\text{proxAnguloStack}) * \sin(\text{proxAnguloFatia}), raio * \sin(\text{proxAnguloStack}), raio * \cos(\text{proxAnguloStack}) * \cos(\text{proxAnguloFatia}))$
- Ponto 3 : $(raio * \cos(\text{proxAnguloStack}) * \sin(\text{anguloFatia}), raio * \sin(\text{proxAnguloStack}), raio * \cos(\text{proxAnguloStack}) * \cos(\text{anguloFatia}))$

Assim para as **camadas intermédias** os pontos dos triângulos serão os seguintes:

- **Triângulo 1**

- Ponto 1 : $((raio * \cos(\text{anguloStack}) * \sin(\text{anguloFatia}), raio * \sin(\text{anguloStack}), raio * \cos(\text{anguloStack}) * \cos(\text{anguloFatia}))$
- Ponto 2 : $((raio * \cos(\text{anguloStack}) * \sin(\text{proxAnguloFatia}), raio * \sin(\text{anguloStack}), raio * \cos(\text{anguloStack}) * \cos(\text{proxAnguloFatia}))$
- Ponto 3 : $((raio * \cos(\text{proxAnguloStack}) * \sin(\text{anguloFatia}), raio * \sin(\text{proxAnguloStack}), raio * \cos(\text{proxAnguloStack}) * \cos(\text{anguloFatia}))$

- **Triângulo 2**

- Ponto 1 : $(raio * \cos(\text{anguloStack}) * \sin(\text{proxAnguloFatia}), raio * \sin(\text{anguloStack}), raio * \cos(\text{anguloStack}) * \cos(\text{proxAnguloFatia}))$
- Ponto 2 : $(raio * \cos(\text{proxAnguloStack}) * \sin(\text{proxAnguloFatia}), raio * \sin(\text{proxAnguloStack}), raio * \cos(\text{proxAnguloStack}) * \cos(\text{proxAnguloFatia}))$
- Ponto 3 : $(raio * \cos(\text{proxAnguloStack}) * \sin(\text{anguloFatia}), raio * \sin(\text{proxAnguloStack}), raio * \cos(\text{proxAnguloStack}) * \cos(\text{anguloFatia}))$

3.4 Cone

A construção do **Cone** é muito semelhante á construção da esfera. No entanto na esfera, ao nível das stacks trabalhamos mais com ângulos, e no cone trabalhamos muito mais com alturas e raios. No que toca a slices, o comportamento é análogo.

Assim, para construirmos um Cone, recebemos o raio da sua base, a altura do mesmo, o seu número de slices (divisões verticais) e o seu número de stacks (divisões horizontais). Temos neste momento as seguintes informações:

- Cada slice possui um angulo default de: $(2 * M_PI) / nrSlices$
- Cada stack possui uma altura de: $alturaCone / nrStacks$
- Cada mudança de raio para a stack seguinte é de: $raioBase / nrStacks$

Assim, cada cone é desenhado uma slice de cada vez. Este processo ocorre um certo número de vezes que equivala o número de slices pedidas pelo Utilizador. Dentro do desenha de cada slice, o processo é dividido em 3 fases (1 para desenhar o triângulo da base e outras 2 para desenhar os vários níveis das stack). Sendo assim, quando iniciamos o desenho de uma slice temos sempre:

$$float\ angulo = fatiaNr * defaultAngleFatia;$$
$$float\ proxAngulo = angulo + defaultAngleFatia;$$
$$float\ alturaAtual = 0;$$
$$float\ proxAltura = alturaStack;$$
$$float\ raioAtual = raioBase;$$
$$float\ proxRaio = raioBase - mudancaRaioStack;$$

Entenda-se aqui que fatiaNr é a fatia onde nos encontramos atualmente. Este fatiaNr é incrementado ao fim de construirmos todos os pontos de uma slice. Isto acontece recorrendo a um for do tipo:

$$for\ (int\ fatiaNr = 0; fatiaNr < nrSlices; fatiaNr++)$$

Após isso é desenhado o triângulo que compõe a base:

- `Ponto(0.0f, 0.0f, 0.0f);`
 `Ponto(raioBase * cos(angulo), 0.0f, raioBase * sin(angulo));`
 `Ponto(raioBase * cos(proxAngulo), 0.0f, raioBase * sin(proxAngulo));`

Seguidamente e através do seguinte `for`

`for (int nrCamada = 1; nrCamada < nrStacks; nrCamada++)`

são desenhados triângulos que compõe a lateral do cone. Importante realçar que se o cone apenas tiver 1 stack, este processo não acontece e não entramos neste loop. Caso contrário em cada loop construímos a stack em estudo da slice que estamos a tratar. Este processo irá conter um número de iterações igual a `nrStacks - 1`. Assim para cada iteração iremos construir 2 triângulos que serão:

- `Ponto(proxRaio * cos(proxAngulo), proxAltura, proxRaio * sin(proxAngulo));`
 `Ponto(raioAtual * cos(proxAngulo), alturaAtual, raioAtual * sin(proxAngulo));`
 `Ponto(raioAtual * cos(angulo), alturaAtual, raioAtual * sin(angulo));`
- `Ponto(proxRaio * cos(angulo), proxAltura, proxRaio * sin(angulo))`
 `Ponto(proxRaio * cos(proxAngulo), proxAltura, proxRaio * sin(proxAngulo));`
 `Ponto(raioAtual * cos(angulo), alturaAtual, raioAtual * sin(angulo));`
- No final de cada iteração os seguintes valores são atualizados:
 - » `alturaAtual += alturaStack;`
 - » `proxAltura += alturaStack;`
 - » `raioAtual -= mudancaRaioStack;`
 - » `proxRaio -= mudancaRaioStack;`

No final é desenhada a última stack, que possui apenas um triângulo (ou no caso de só querermos uma stack, é o único passo que realizamos além do do triângulo da base). Optamos por abordar desta forma o problema para não haver desperdício de pontos e de memória (no entanto, este processo poderia ser feito dentro do loop interior, no entanto, iriam ser criados dois triângulos um cima do outro escusadamente, desperdiçando recursos).

Assim, o último passa será construir a ultima stack e passará por fazer a construção dos seguintes pontos:

- `Ponto(raioAtual * cos(proxAngulo), alturaAtual, raioAtual * sin(proxAngulo));`
- `Ponto(raioAtual * cos(angulo), alturaAtual, raioAtual * sin(angulo));`
- `Ponto(0.0f, alturaCone, 0.0f);`

Este processo todo é, como referido anteriormente, repetido `nrSlices` vezes, sendo `nrSlices` o número de slices que o Utilizador decide que o cone tenha.

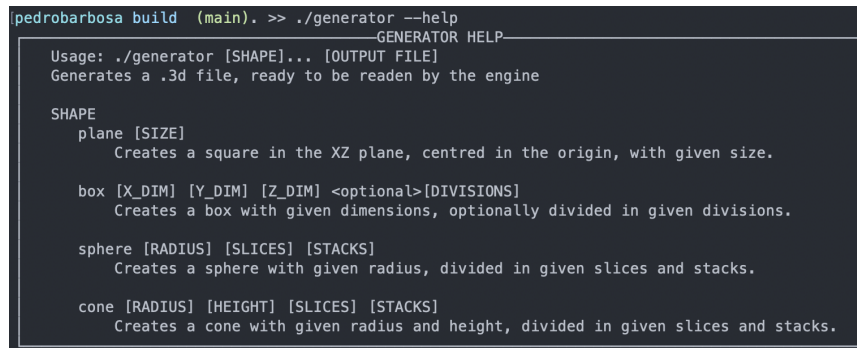
4 Generator

4.1 Descrição

Tal como referido anteriormente, o **Generator** ou **Gerador**, é responsável, tal como o próprio nome indica, por gerar todas as primitivas gráficas pedidas no trabalho prático. Para tal acontecer, este calcula todos os vértices necessários para formar os diferentes triângulos que, como estudado nesta unidade curricular, são a unidade de construção das diferentes primitivas gráficas. De seguida, guarda os diferentes pontos criados num determinado ficheiro pedido pelo utilizador.

4.2 Funcionalidades

De forma a ser mais perceptível e intuitivo quais são as diversas funcionalidades do Generator, o nosso grupo decidiu criar um menu de ajuda com as mesmas. Este menu pode ser obtido através do seguinte comando: `./generator -help`.



```
pedrobarbosa build (main). >> ./generator --help
                                GENERATOR HELP
Usage: ./generator [SHAPE]... [OUTPUT FILE]
Generates a .3d file, ready to be readen by the engine

SHAPE
  plane [SIZE]
    Creates a square in the XZ plane, centred in the origin, with given size.

  box [X_DIM] [Y_DIM] [Z_DIM] <optional>[DIVISIONS]
    Creates a box with given dimensions, optionally divided in given divisions.

  sphere [RADIUS] [SLICES] [STACKS]
    Creates a sphere with given radius, divided in given slices and stacks.

  cone [RADIUS] [HEIGHT] [SLICES] [STACKS]
    Creates a cone with given radius and height, divided in given slices and stacks.
```

Figura 1: Menu de Ajuda do Generator

4.3 Demonstração

Tal como demonstrado no menu de ajuda do *Generator* acima apresentado, para se gerar uma determinada primitiva geométrica deve ser passado como input o nome da mesma acompanhado das respetivas dimensões e divisões, caso seja necessário, e também do nome do ficheiro onde queremos guardar o output.

- **Plano**

plane [SIZE] [OUTPUT FILE]

- **Box**

box [XDIM] [YDIM] [ZDIM] <optional>[DIVISIONS] [OUTPUT FILE]

- **Esfera**

sphere [RADIUS] [SLICES] [STACKS] [OUTPUT FILE]

- **Cone**

cone [RADIUS] [HEIGHT] [SLICES] [STACKS] [OUTPUT FILE]

Desta forma, o exemplo de um possível input seria:

```
./generator box 2 2 2 3 box.3d
```

Figura 2: Exemplo de um Input para o Generator

Depois de executado este comando, o *generator* cria, em caso de inexistência, uma diretoria *files3D/* onde serão guardados todos os ficheiros output. Estes ficheiros irão conter na linha inicial o número total de pontos utilizados para gerar a primitiva geométrica. De seguida, seguem-se linhas formadas pelas coordenadas x , y e z , que são floats, separados por vírgulas e terminam com um $\backslash n$.

```
[pedrobarbosa files3D (main). >> cat plane.3d
6
2.000000, 0.000000, 2.000000
2.000000, 0.000000, -2.000000
-2.000000, 0.000000, -2.000000
-2.000000, 0.000000, 2.000000
-2.000000, 0.000000, 2.000000
2.000000, 0.000000, 2.000000
```

Figura 3: Exemplo de um Output do Generator

5 Engine

5.1 Descrição

Tal como descrito na estrutura do projeto, o *Engine* ou *Motor*, é responsável por interpretar e representar os ficheiros *XML* que lhe são passados como argumentos. Por sua vez, os ficheiros *XML* contêm as indicações dos ficheiros que o *Engine* deverá carregar de forma a representar graficamente no ecrã as diferentes primitivas geométricas.

5.2 Funcionalidades

De forma a ser mais perceptível e intuitivo quais são as diversas funcionalidades do Engine o nosso grupo decidiu criar um menu de ajuda com as mesmas. Este menu pode ser obtido através do seguinte comando `./engine -help`.

```
pedrobarbosa build (main). >> ./engine --help
ENGINE HELP
Usage: ./engine [XML FILE]
Displays all primitives loaded from XML FILE

Camera options
  a : Moves camera to the left
  d : Moves camera to the right
  w : Moves camera up
  s : Moves camera down
  e : Zoom in
  q : Zoom out

Scene options
  t : Cycle between drawing modes
  c : Cycle between white and random colors
  1-9 : Draw a single object
  0 : Draw all objects

Press ESC at any time to exit program
```

Figura 4: Menu de Ajuda do Engine

5.3 Demonstração

O *Engine* é então responsável por ler um ficheiro de configuração *XML* e apresentar graficamente as diferentes primitivas geométricas. Realçamos que os ficheiros presentes nestes ficheiros *XML* devem ter sido anteriormente carregados pelo *Generator*. Após todo este processo é possível interagir com os modelos gerados através dos comandos que foram apresentados acima no menu de ajuda.

O exemplo de um ficheiro *XML* é o seguinte:

```
[pedrobarbosa filesXML (main). >> cat all.xml
<scene>
  <model folder = "../..../files3D/" />
  <model file = "plane.3d" />
  <model file = "box.3d" />
  <model file = "sphere.3d" />
  <model file = "cone.3d" />
</scene>
```

Figura 5: Exemplo de um Ficheiro XML

Desta forma o exemplo de um possível input seria:

```
[pedrobarbosa build (main). >> ./engine all.xml
```

Figura 6: Exemplo de um Input para o Engine

6 Modelos 3D

6.1 Plano

Plano gerado com dimensão 4. Cria dois triângulos paralelos ao eixo XZ e centrados na origem.

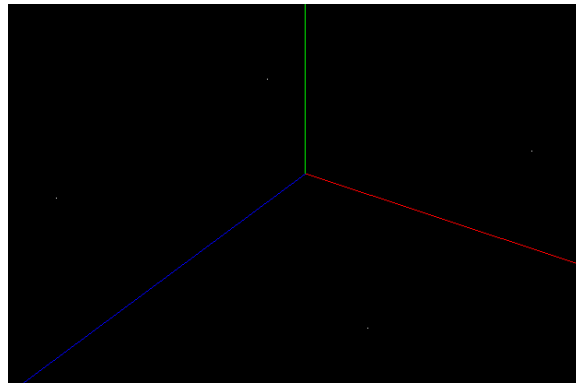


Figura 7: Apenas Pontos

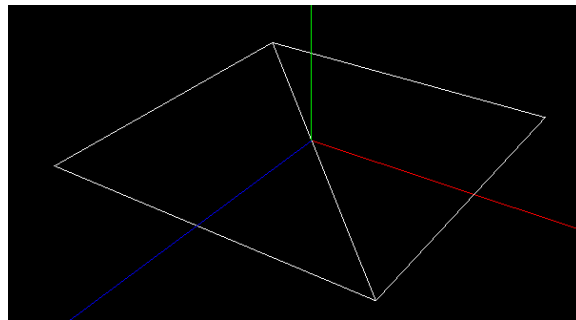


Figura 8: Apenas Linhas

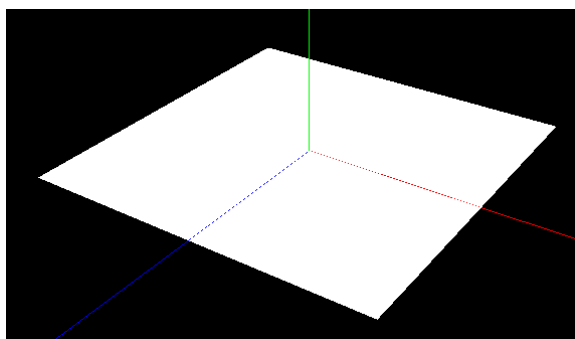


Figura 9: Preenchido

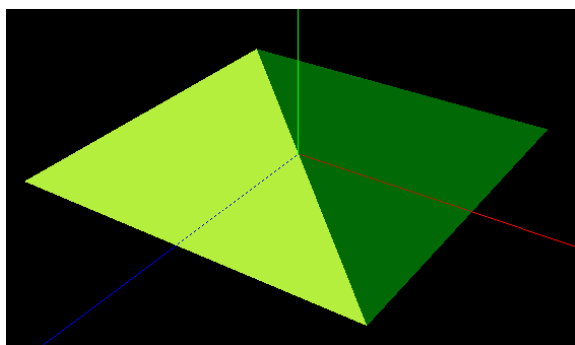


Figura 10: Cores Aleatórias

6.2 Box

Box gerada com as dimensões 3, 3, 3 (dimensões em x,y e z respectivamente) e com 3 divisões (opcional).

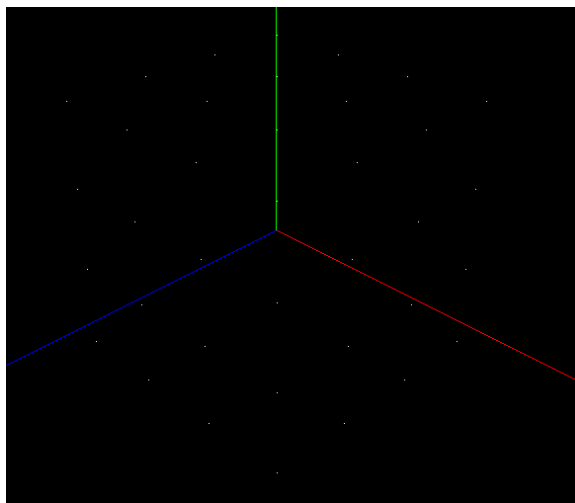


Figura 11: Apenas Pontos

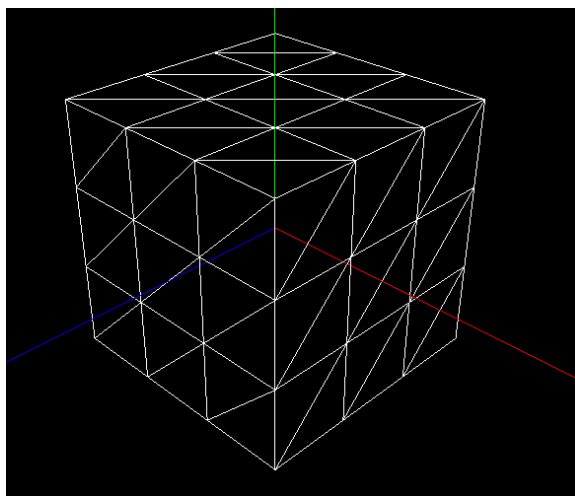


Figura 12: Apenas Linhas

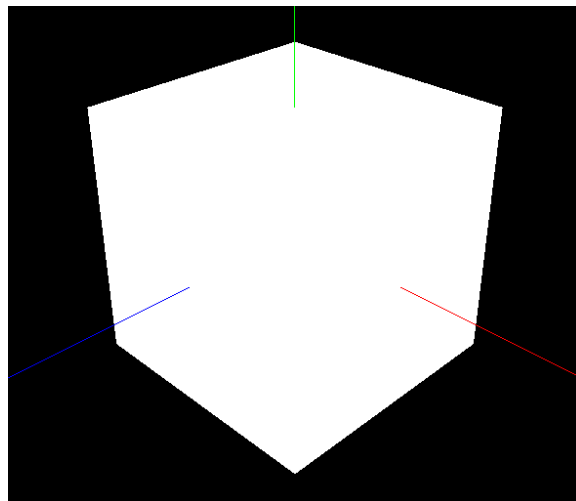


Figura 13: Preenchido

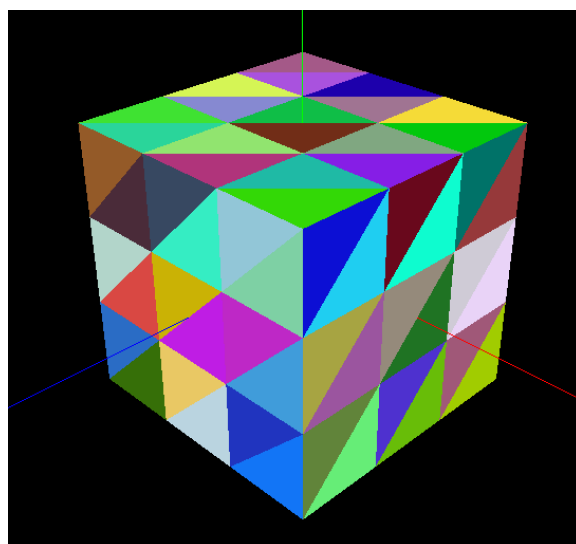


Figura 14: Cores Aleatórias

6.3 Esfera

Esfera gerada com as dimensões dadas sendo estas raio 2, e 50 fatias e camadas.

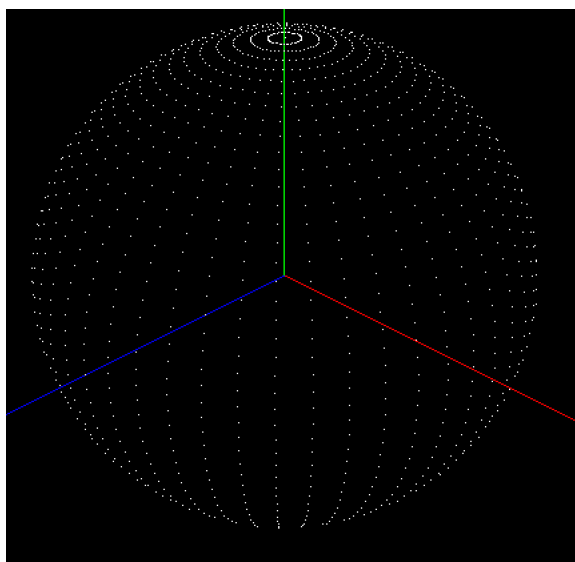


Figura 15: Apenas Pontos

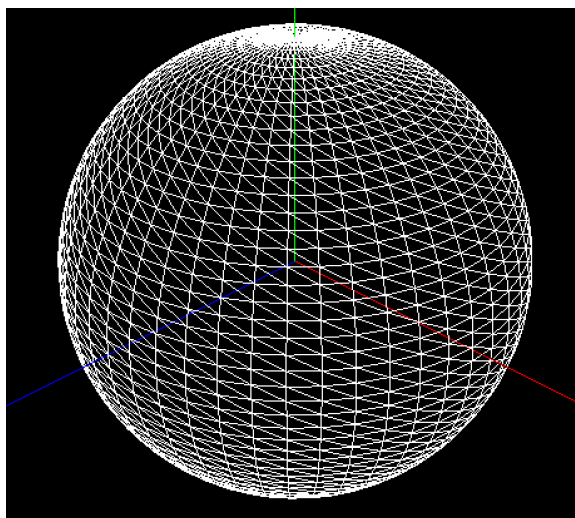


Figura 16: Apenas Linhas

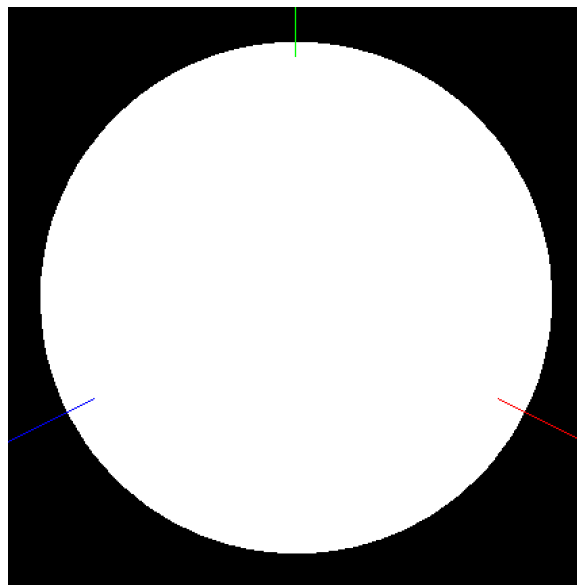


Figura 17: Preenchido

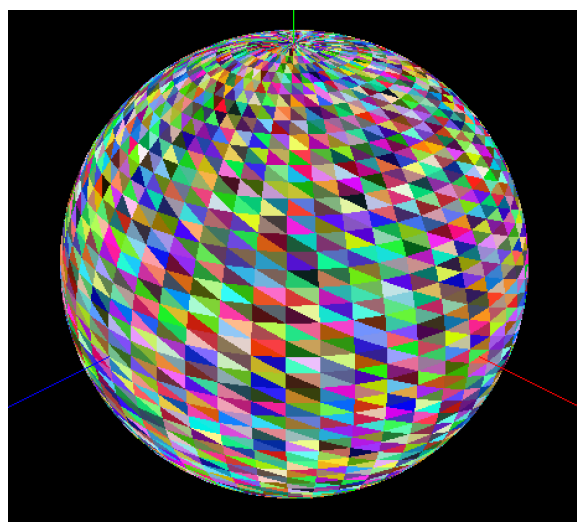


Figura 18: Cores Aleatórias

6.4 Cone

Cone gerado com as dimensões dadas sendo estas raio 1, altura 2 e 10 fatias e camadas.

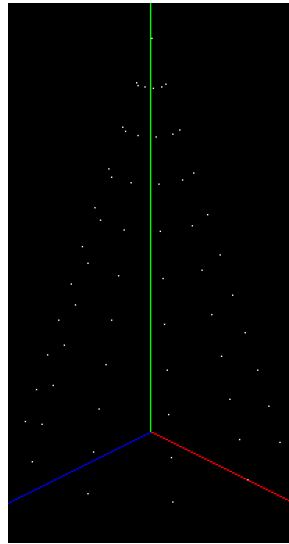


Figura 19: Apenas Pontos

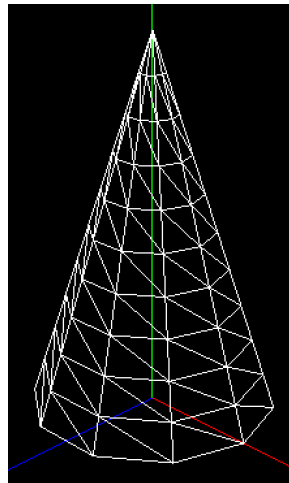


Figura 20: Apenas Linhas

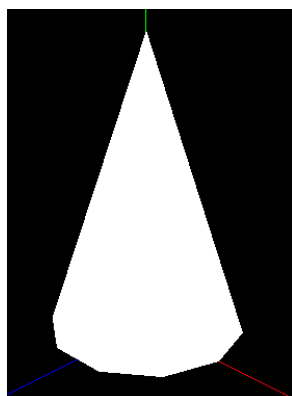


Figura 21: Preenchido

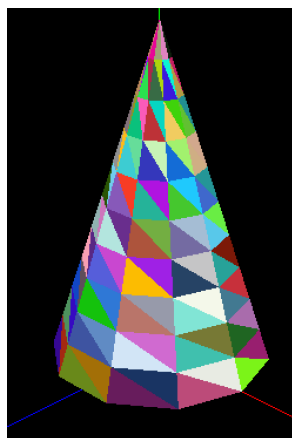


Figura 22: Cores Aleatórias

7 Conclusão e Trabalho Futuro

Concluindo, todo o trabalho realizado nesta primeira fase do trabalho prático foi muito importante, na medida em que nos permitiu trabalhar com ferramentas e conceitos relativos à unidade curricular de Computação Gráfica e, deste modo, ganhar alguma experiência, por exemplo, em *GLUT* e *OpenGL*.

Em simultâneo, como todo o trabalho foi realizado na linguagem C++, a qual nenhum dos elementos do grupo tinha tido qualquer contacto anteriormente, permitiu-nos também aumentar o nosso conhecimento em termos de linguagens de programação e a sua utilização.

Esperamos assim, que esta primeira fase sirva como uma boa base para o resto do trabalho prático.

Todo o trabalho realizado nesta primeira fase foi bastante importante, uma vez que nos permitiu acumular uma experiência relativa à utilização e especialização em ferramentas e conceitos relativos à computação gráfica, utilizando neste âmbito OpenGL e Glut. Em paralelo também em nós foi permitido aprender uma linguagem ainda não abordada neste curso, C++, que foi bastante amigável em termos de implementação de todo o processo.