

Universidade do Minho
Licenciatura em Engenharia Informática

Computação Gráfica

Fase 1 - Grupo 20

André Silva - A87958 Armando Silva - A87949
Joana Oliveira - A87956 João Nunes - A87972

Março 2022



Conteúdo

1	Introdução	3
2	Estrutura do projeto	4
3	Generator/Gerador	5
3.1	Primitivas	5
3.1.1	Plano	5
3.1.2	Caixa	6
3.1.3	Esfera	6
3.1.4	Cone	7
3.1.5	Toro - Extra	9
3.2	Criação ficheiro .3d	10
4	Engine/Motor	11
4.1	Leitura de ficheiros	11
4.1.1	XML	11
4.1.2	3D	11
4.2	Desenho das figuras	12
4.3	Câmara	12
4.4	Comandos	12
5	Demos	14
6	Conclusão	17

Lista de Figuras

1	Diagrama de <i>packages</i>	4
2	Ponto numa esfera utilizando os ângulos de slices e stacks	6
3	Coordenadas cartesianas da base	7
4	Divisão por slices	7
5	Representação de um cone através de slices e stacks	8
6	Coordenadas cartesianas do toro	9
7	Estrutura de um ficheiro .3d	10
8	Visualização do cone e da esfera individualmente	14
9	Visualização da junção das duas primitivas	14
10	Visualização da figura em diferentes perspetivas	15
11	Visualização das linhas dos triângulos da figura	15
12	Visualização das figura final da demo 3	16
13	Visualização das linhas da figura final da demo 4	16

1 Introdução

Na primeira fase do projeto é proposto o desenvolvimento de um gerador de vértices e de um motor capaz de desenhar os vértices gerados.

O gerador, de acordo com alguns parâmetros, consegue gerar vértices para quatro diferentes primitivas: o plano (*plane*), a caixa (*box*), a esfera (*sphere*) e o cone (*cone*). Por sua vez, estes vértices são depois guardados em ficheiros .3d. Já o motor, é capaz de ler ficheiros .xml/.3d e apresentar, desenhando, as figuras, com os vértices anteriormente gerados, graficamente.

Para completar esta fase, o grupo decidiu adicionar a opção de geração de mais uma primitiva, o toro (*torus*). E ainda, a elaboração de uma câmara, a coloração dos desenhos, a possibilidade de alterar o modo do desenho, entre outras que vão ser explicadas ao longo deste relatório.

2 Estrutura do projeto

Para obtermos uma melhor organização do projeto, decidimos estruturá-lo recorrendo a diretorias separadas, uma para o Gerador (*Generator*), uma para o Motor (*Engine*), outra para os Modelos (*Models*) e uma última para o *tinyxml2*. Apresentamos a seguir, um diagrama de *packages* que melhor representa a nossa estruturação.

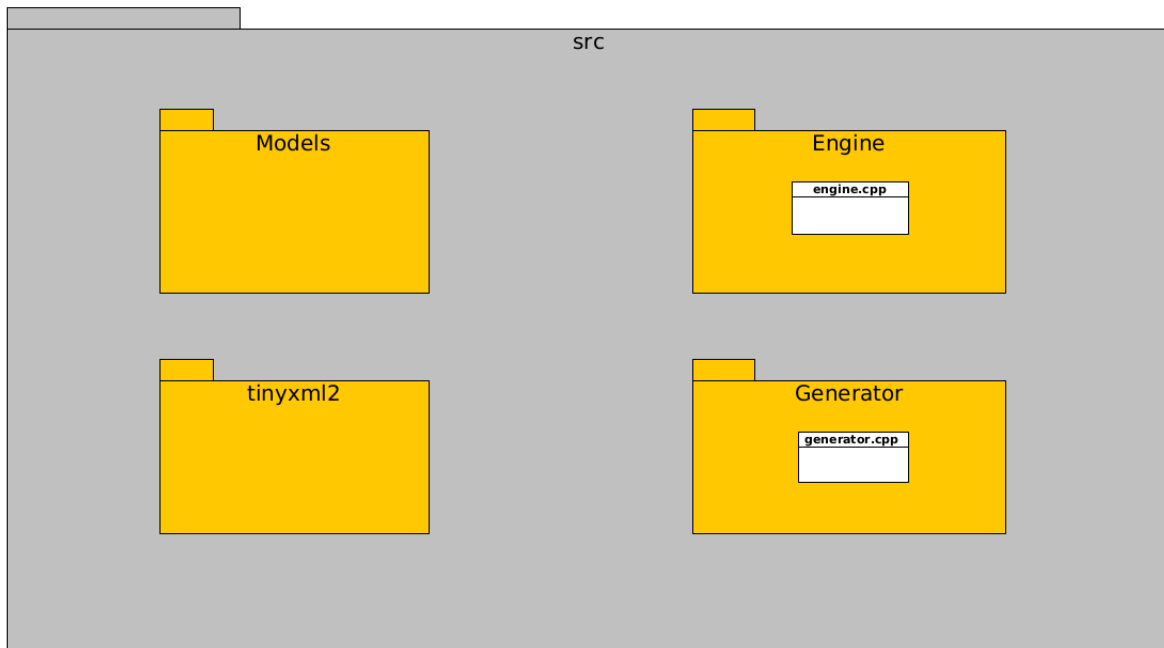


Figura 1. Diagrama de *packages*

O *package* **Generator** possui todo o código referente à geração de todos os vértices necessários para a representação das várias figuras implementadas. A criação dos ficheiros .3d também é feita nesta diretoria, estes que vão ser guardados em *Models* juntamente com todos os ficheiros .xml recebidos.

No **Engine** temos todo o código associado à leitura dos ficheiros .xml e .3d. O desenho de todas as figuras geradas e todos os comandos disponibilizados durante a visualização das primitivas também está implementado nesta diretoria. Este *package* utiliza o *tinyxml2* para facilitar a leitura dos ficheiros .xml.

Já em **Models** temos todos os ficheiros das figuras com extensão .3d e os de configuração em .xml.

3 Generator/Gerador

Aplicação responsável pela geração dos vértices que irão formar todas as primitivas implementadas. É ainda responsável pela criação dos ficheiros .3d com os vértices gerados. A seguir, explicamos todos os passos necessários para a obtenção de cada uma das primitivas e também o procedimento para a criação dos respetivos ficheiros .3d.

3.1 Primitivas

Neste projeto foi pedido a implementação de várias primitivas: *plane*, *box*, *sphere* e *cone*. Foi decidido também, implementar a primitiva *torus* como primitiva adicional.

De modo a gerar estas primitivas é necessário dar as informações específicas para a criação das mesmas. Os itens apresentados de seguida representam as informações necessárias para cada primitiva:

- **plane:** length, divisions;
- **box:** dimension, divisions;
- **sphere:** radius, slices, stacks;
- **cone:** radius, height, slices, stacks;
- **torus:** radius, ringRadius, slices, stacks.

Todas as funções a seguir referidas retornam uma string com os pontos necessários divididos por "\n", e que contém no seu início o número total de pontos que são criados. Esta string é depois inserida nos ficheiros .3d da primitiva construída.

3.1.1 Plano

Para a criação do plano foi desenvolvida a função **createPlane**, que recebe os parâmetros *length* e *divisions*. Esta função permite a criação de planos em todos os planos possíveis, tendo sido por isso necessário atribuir a cada um deles um ID identificativo.

Para construir todos os triângulos necessários de acordo com o número de divisões indicado, é então calculado o tamanho de cada quadrado constituinte da grelha, através da divisão do tamanho total de cada lado pelo número de divisões. Assim vão sendo construídos os vários quadrados, utilizando como referência para as suas coordenadas, a distância calculada anteriormente.

3.1.2 Caixa

Na criação da caixa, é utilizada a função **createBox** que recebe como argumento as *dimensions* e *divisions*.

O cálculo do número total de pontos necessários é feito através da seguinte expressão:

$$N = 6^2 * divisions^2$$

Onde 6 representa o número de vértices de um quadrado (3 + 3 vértices do triângulo).

Para desenhar cada face da caixa utilizamos a função auxiliar **createPlane** que nos ajuda a criar todas as faces de cada plano que constituem a caixa.

3.1.3 Esfera

Para o desenho da esfera foi criada a função **createSphere**, que recebe como argumentos o *radius*, as *slices* e as *stacks*. As *slices* referem-se às divisões verticais da esfera e as *stacks* às divisões horizontais. Tendo isto em conta, é então possível calcular os intervalos em que se encontram os seguintes ângulos.

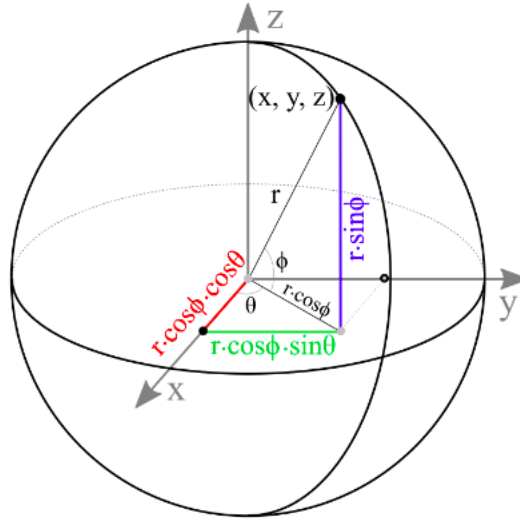


Figura 2. Ponto numa esfera utilizando os ângulos de slices e stacks

$$\begin{aligned} -\frac{\pi}{2} &\leq \phi \leq \frac{\pi}{2} \\ 0 &\leq \theta \leq 2\pi \end{aligned}$$

Assim sendo, temos então o primeiro ciclo for que irá percorrer todas as stacks utilizando como limites os seguintes valores de coordenadas y, sendo $\beta = \frac{-\pi}{2} + i * \phi$ e i o índice que percorre o ciclo.

$$\begin{aligned} y1 &= r * \sin(\beta) & (inferior) \\ y2 &= r * \sin(\phi + \beta) & (superior) \end{aligned}$$

De seguida temos então dentro desse ciclo, outro ciclo for que irá percorrer todas as slices e criar os 4 pontos necessários para criar cada secção. Estes serão calculados tendo em conta as seguintes expressões:

$$\begin{aligned} x &= r * \cos\phi * \sin\theta \\ z &= r * \cos\phi * \cos\theta \end{aligned}$$

O cálculo do número total de pontos da figura é feito através da seguinte expressão:

$$N = \text{slices} * \text{stacks} * 6$$

3.1.4 Cone

No desenho do cone foi implementada a função **createCone**, que recebe como parâmetro o *radius*, a *height*, as *slices* e *stacks*. A contagem dos pontos é feita incrementando o seu número à medida que os triângulos são criados.

Numa primeira fase, é criada a base usando coordenadas polares e seguindo as seguintes expressões:

$$\begin{aligned} &\textit{Cartesian Coordinates} \\ px &= r * \sin(\alpha); \\ pz &= r * \cos(\alpha); \end{aligned}$$

Figura 3. Coordenadas cartesianas da base

onde α é determinado da seguinte forma:

$$\alpha = \frac{2*\pi}{\text{slices}}$$

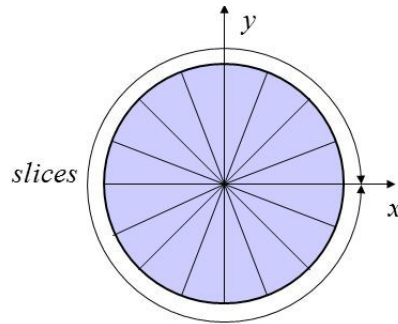


Figura 4. Divisão por slices

Para o cálculo da lateral é utilizado um ângulo β que é calculado da seguinte forma:

$$\beta = \frac{height}{stacks}$$

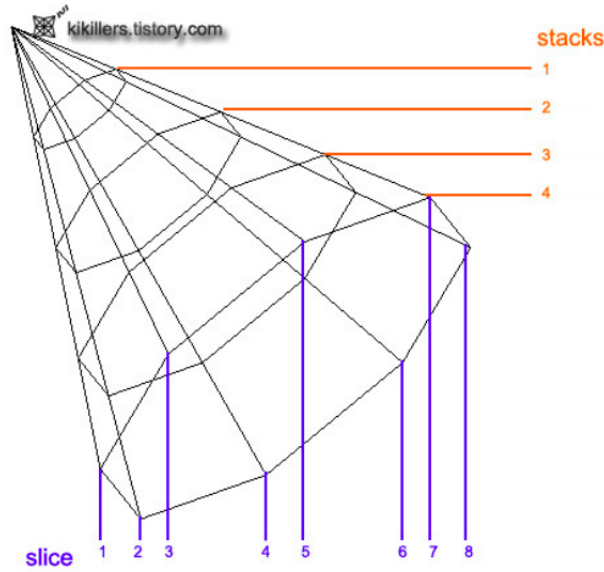


Figura 5. Representação de um cone através de slices e stacks

De seguida é utilizado um ciclo for que irá percorrer todas as stacks. Uma vez que cada stack terá que ter dois raios diferentes, o inferior e o superior, estes serão calculados tendo em conta o número de stacks já existentes. Sendo $\theta = i * \beta$ e i o índice do ciclo, temos então:

$$\begin{aligned} r1 &= r * (i * \frac{radius}{stacks}) && (inferior) \\ r2 &= r * ((i+1) * \frac{radius}{stacks}) && (superior) \end{aligned}$$

Teremos então outro ciclo for que permitirá percorrer todas as stacks e, à semelhança da esfera, criar os 4 pontos necessários. De maneira a conseguirmos criá-los de maneira circular, os pontos utilizaram todas as expressões do tipo:

$$\begin{aligned} x &= r * \sin\alpha \\ z &= r * \cos\alpha \end{aligned}$$

3.1.5 Toro - Extra

Para a implementação desta figura extra foi criada a função **createTorus**, que recebe como função o *radius*, *ringRadius*, *slices* e *stacks*. O número total de vértices é dado pela expressão:

$$N = slices * stacks * 6$$

As coordenadas são dadas da seguinte forma:

$$x = (radius + ringRadius * \cos(\alpha)) * \cos(\beta)$$

$$y = ringRadius * \sin(\alpha)$$

$$z = (radius + ringRadius * \cos(\alpha)) * \sin(\beta)$$

onde α e β são calculados seguindo as seguintes expressões:

$$\alpha = \frac{2*\pi}{slices}$$

$$\beta = \frac{2*\pi}{stacks}$$

De forma a tornar mais clara a interpretação destes cálculos apresentaremos a seguinte imagem:

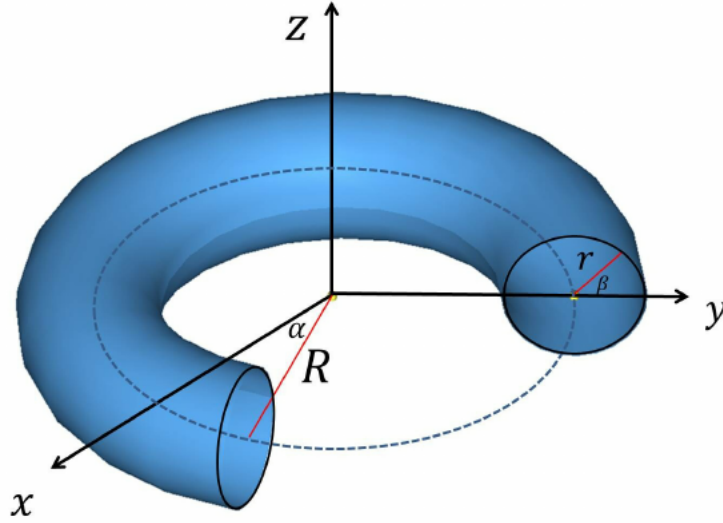
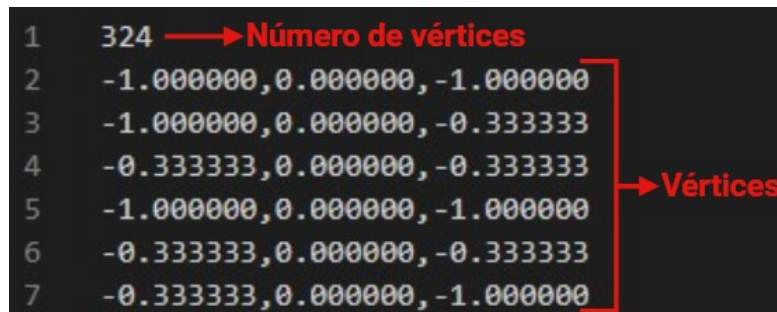


Figura 6. Coordenadas cartesianas do toro

3.2 Criação ficheiro .3d

Depois de todos os vértices terem sido gerados, estes vão ser escritos no ficheiro dado no início do programa. Caso o ficheiro já exista, o seu conteúdo é descartado e os novos vértices serão escritos normalmente. Caso este não exista, vai ser criado.

A estrutura destes ficheiros é igual para todas as primitivas, sendo que a primeira linha vai sempre ser o número total de vértices gerados e as restantes vão conter todos os vértices, um por linha, separados por vírgulas, ou seja, 'x,y,z'.



The image shows a text-based representation of a .3d file. The first line contains the number '324' followed by a red arrow pointing to the text 'Número de vértices'. The subsequent lines (2 through 7) contain three floating-point numbers separated by commas, representing the x, y, and z coordinates of vertices. A red bracket on the right side of these lines points to the text 'Vértices'.

Line	Content
1	324 → Número de vértices
2	-1.000000,0.000000,-1.000000
3	-1.000000,0.000000,-0.333333
4	-0.333333,0.000000,-0.333333
5	-1.000000,0.000000,-1.000000
6	-0.333333,0.000000,-0.333333
7	-0.333333,0.000000,-1.000000

Figura 7. Estrutura de um ficheiro .3d

4 Engine/Motor

Aplicação responsável pela leitura de ficheiros .xml e .3d, e o desenho das figuras contidos nesses mesmos ficheiros. É ainda responsável pela representação dos eventos implementados, estes que serão invocados pelo utilizador ao clicar em certas teclas. Apresentamos e explicamos a seguir, as partes mais importantes desta aplicação.

4.1 Leitura de ficheiros

São lidos dois tipos de ficheiros diferentes, um contém todas as configurações necessárias para o posicionamento da câmara e quais as figuras que necessitam de ser desenhadas, já o outro vai conter todos os pontos precisos para desenhar uma primitiva, sendo estes, .xml e .3d, respetivamente.

Explicamos a seguir, o procedimento da aplicação para a leitura de cada um destes tipos de ficheiro.

4.1.1 XML

Para ler o ficheiro .xml pretendido utilizamos a biblioteca *tinyxml2*, que transforma os conteúdos do ficheiro em objetos que são de seguida lidos e processados através de funções existentes nesta mesma biblioteca. Deste ficheiro .xml retiramos informações sobre o posicionamento da câmara e também quais os ficheiros .3d existentes. Estas informações são guardadas em variáveis globais de modo a serem facilmente acedidas noutros locais do *engine*.

4.1.2 3D

Depois do ficheiro de configuração ter sido lido e a informação de quais os modelos necessários para o desenho da primitiva ter sido guardada, vão ser armazenados, um a um, os pontos dos modelos requisitados. Para tal, é aberto o ficheiro .3d e é guardado o número total de vértices presentes no ficheiro (primeira linha do ficheiro .3d). De seguida, até esse número ser atingido, todos os pontos são introduzidos numa *struct Vertice*, sendo estes colocados num *vector*. Quando o número total é alcançado, esse *vector* é armazenado num outro *vector* global, este que vai conter todas os desenhos possíveis.

Com todos os ficheiros lidos (.xml e .3d), a aplicação vai passar para o desenho das figuras, parte explicada na subsecção a seguir.

4.2 Desenho das figuras

O desenho das figuras começa pela apresentação dos três eixos, X a vermelho, Y a verde e Z a azul. Passa depois para o desenho da figura, que dependendo de qual está a ser pedida, a aplicação vai percorrer um certo índice do *vector* global, falado na subsecção acima e desenhar de três em três vértices (um triângulo). De salientar, que caso o ficheiro .xml contenha mais do que uma primitiva, a aplicação vai desenhar cada uma individualmente, dependendo de qual o utilizador está a observar. Na secção seguinte, vão ser apresentados alguns comandos que ajudam a obter todas as primitivas pedidas.

A coloração é também feita aqui, pois para cada vértice é escolhida aleatoriamente uma cor, permitindo assim um triângulo ter três cores distintas para cada um dos seus vértices, isto origina um gradiente entre as cores dentro de cada triângulo.

4.3 Câmara

Como funcionalidade extra foi implementada uma câmara de exploração, obedecendo às coordenadas cartesianas a seguir demonstradas:

$$\begin{aligned}x &= raio * \cos(\beta) * \sin(\alpha) \\ y &= raio * \sin(\beta) \\ z &= raio * \cos(\beta) * \cos(\alpha)\end{aligned}$$

onde α corresponde ao ângulo horizontal e β corresponde ao ângulo vertical.

De forma a suavizar a alteração da câmara estática para a câmara de exploração, é necessário descobrir os valores dos ângulos α e β e também o seu raio, i.e, a distância da câmara até à origem. Para isso utilizamos as seguintes equações, sabendo que a câmara se encontra nas coordenadas (x,y,z):

$$\begin{aligned}r &= \sqrt{x^2 + y^2 + z^2} \\ \beta &= \arcsin\left(\frac{y}{r}\right) \\ \alpha &= \arccos\left(\frac{x}{\cos(\beta)}\right)\end{aligned}$$

Como nota final, de forma a respeitar a regra da mão direita no plano, a construção dos triângulos é invertida quando a câmara se encontra na parte negativa do eixo do y. No que toca à manipulação da câmara, esta será explicada na secção seguinte.

4.4 Comandos

Apresentamos agora todos os comandos implementados que um utilizador pode usar durante a visualização das primitivas pretendidas.

Subir a câmara - Teclas W/w

De forma a subir a câmara é necessário aumentar o ângulo vertical, i.e, o ângulo β , este é aumentado pela constante $\frac{\pi}{12}$. De forma a não virar a câmara ao contrário

este ângulo tem um limite de $\frac{\pi}{2}$.

Descer a câmara - Teclas S/s

Segue o mesmo formato que o comando que o comando "subir a câmara", no entanto, o β é incrementado $-\frac{\pi}{12}$ e tem um limite de $-\frac{\pi}{2}$.

Mover a câmara para a esquerda - Teclas A/a

O ângulo que varia a movimentação para esquerda é o α , e este não tem limite. Tem a mesma incrementação que o comando "descer a câmara".

Mover a câmara para a direita - Teclas D/d

Por fim, a movimentação para a direita, faz o oposto do movimento para esquerda, i.e, é incrementado pela constante $\frac{\pi}{12}$. Também não contém limite.

Avançar para a figura seguinte - Teclas E/e

Como dito anteriormente, quando é pedido o desenho de mais do que uma primitiva, a aplicação desenha todas individualmente e a final é a junção de todas as figuras. Para o utilizador conseguir avançar nas figuras, foi implementado este comando que quando é invocado, avança para a próxima figura pedida pelo utilizador.

Para tal, temos uma variável global que controla o número total de primitivas e o número atual da primitiva, assim é possível avançar para a próxima figura, incrementando a variável da primitiva atual.

Recuar para a figura anterior - Teclas Q/q

Caso o utilizador tenha avançado para outra figura, é possível voltar para figuras anteriores utilizando este comando. Assim, o utilizador tem total liberdade para puder observar qualquer primitiva que tenha pedido.

Este comando utiliza as mesmas variáveis que o comando anterior, apenas decrementando a variável que controla a qual figura que está a ser observada no momento.

Ativar/Desativar os três eixos - Teclas Z/z

Comando que ativa ou desativa os três eixos (x, y e z) durante a visualização de qualquer primitiva. Implementado através da criação de uma variável global que controla se os eixos estão ativados ou desativados, dependendo do seu estado os eixos são ou não desenhados.

Mudar o modo de desenho - Teclas X/x

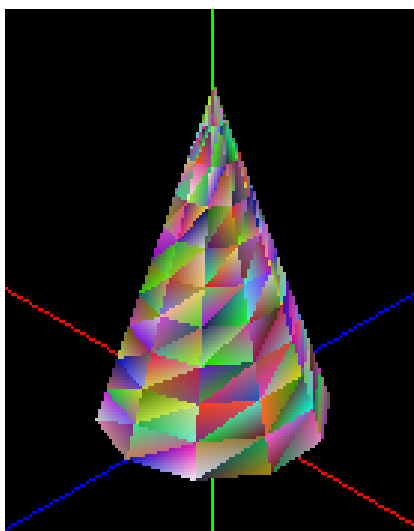
Opção para alterar o modo de desenho das primitivas, tendo as possibilidades de desenho em linha ou preenchido. Isto é, desenhar e colorir apenas as linhas entre os vértices ou preencher todo o espaço entre os triângulos, dando um gradiente de cores entre cada um.

Comando realizado com a criação de uma variável global que controla qual dos modos está a ser utilizado. Dependendo do seu valor, este altera para o outro modo.

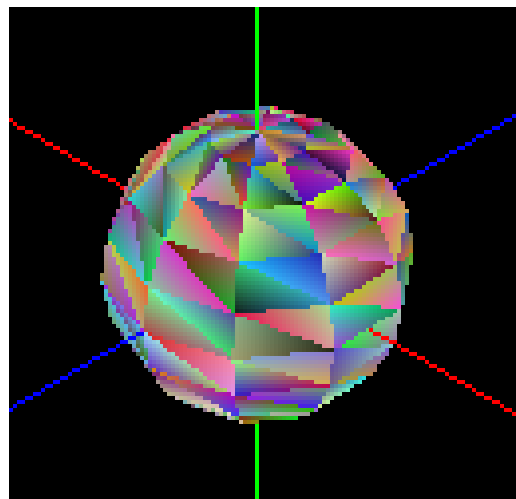
5 Demos

A melhor forma de demonstrarmos tudo o que escrevemos em cima é com alguns *screenshots* das nossas aplicações a executar testes feitos por nós. De salientar, que todos os ficheiros .3d criados para esta demonstração estão na pasta que enviamos com o resto do nosso projeto.

Demo 1: *Cone e Sphere*



(a) Cone



(b) Esfera

Figura 8. Visualização do cone e da esfera individualmente

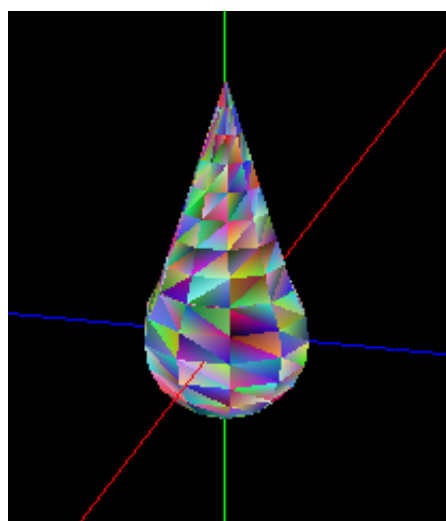
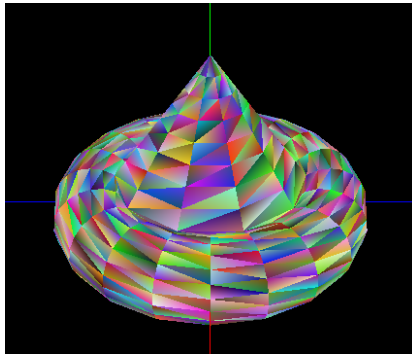
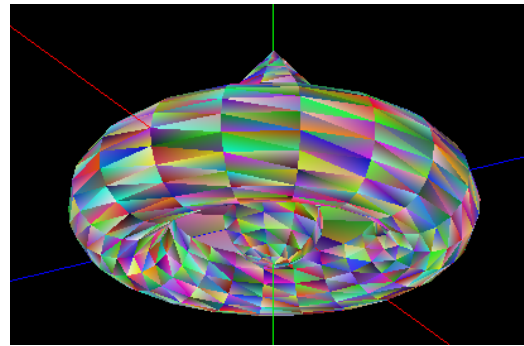


Figura 9. Visualização da junção das duas primitivas

Demo 2: *Cone*, *Sphere* e *Torus*



(a) Perspetiva de cima



(b) Perspetiva de baixo

Figura 10. Visualização da figura em diferentes perspetivas

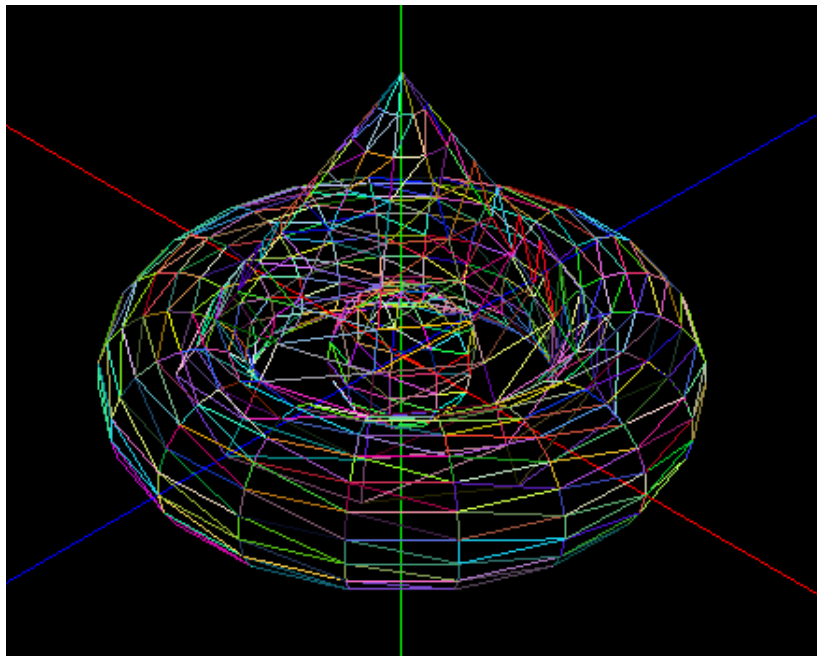


Figura 11. Visualização das linhas dos triângulos da figura

Demo 3: *Box, Cone e Torus*

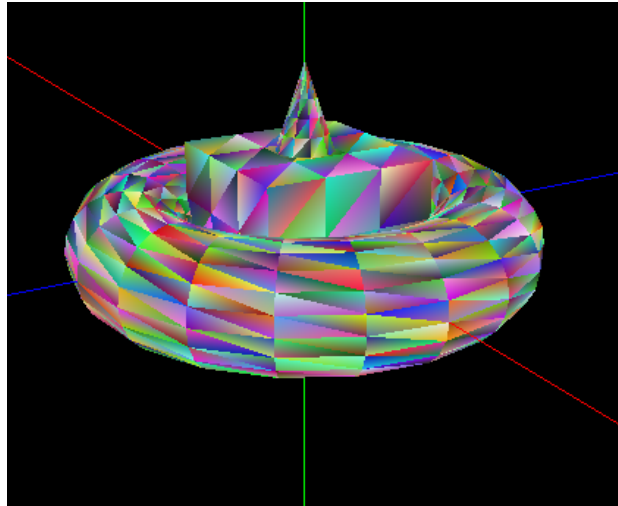


Figura 12. Visualização das figura final da demo 3

Demo 4: *Box, Cone, Sphere e Torus*

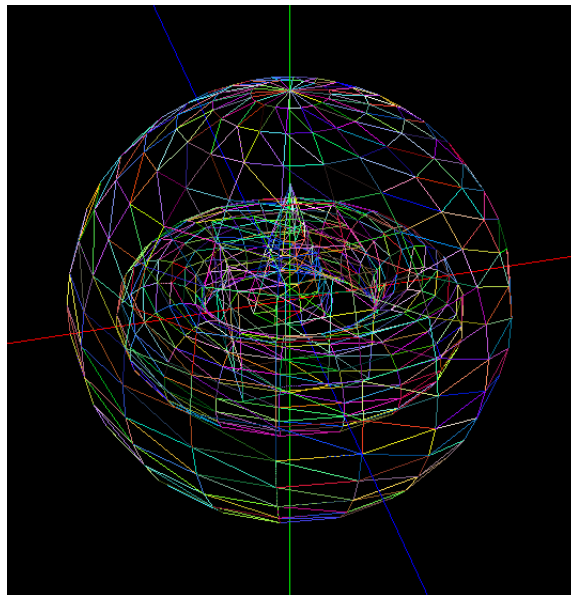


Figura 13. Visualização das linhas da figura final da demo 4

6 Conclusão

Dado por concluída a primeira fase deste projeto, consideramos importante realçar todos os pontos positivos e negativos, e ainda, efetuar uma análise crítica final do trabalho realizado.

O correto desenho de todos os triângulos que formam uma primitiva e a sua agradável colorização são dois aspetos positivos no nosso projeto. A existência de alguns comandos extra oferecem uma liberdade ao utilizador para observar a figura de diferentes formas e feitios, e a junção de todas as figuras pedidas num só desenho permite ao utilizador criar imensos desenhos únicos. Sendo estes, também, tópicos positivos no nosso projeto.

A existência de apenas um tipo de câmara não deixa o utilizador conseguir observar a primitiva como deseja, sendo este um ponto negativo no nosso programa. Apesar de termos adicionado uma primitiva extra, sentimos que poderíamos ter acrescentado mais algumas para expandir as opções na nossa aplicação.

Para concluir, o grupo considera que o trabalho realizado é positivo, dado que cumpre com todos os requisitos propostos, tendo ainda alguns extras para completar esta primeira fase.