Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática
Mestrado Integrado em Engenharia Informática

Unidade Curricular de Computação Gráfica

Ano Letivo de 2023/2024

1ª Fase - Primitivas Gráficas

Diogo Matos
A100741

Júlio Pinto
A100742

Mário Rodrigues
A100109

Miguel Gramoso
A100835

March 08, 2024

CG

Índice

1. Introdução

Este trabalho foi desenvolvido no âmbito da unidade curricular Computação Gráfica (CG) no ano letivo 2023/2024. A primeira fase do projeto consistiu em implementar duas componentes fundamentais ao funcionamento do mesmo, o **generator** e o **engine**.

2. Generator

O **generator** é responsável por produzir o conjunto de pontos para cada primitiva. Todas as primitivas são representadas por triângulos, ou seja, conjuntos de três pontos.

Após serem gerados, os pontos são guardados no ficheiro de saída, que segue a convenção de nomear o arquivo com o nome da primitiva seguido da extensão `.3d`. Por exemplo, no caso de um cone, o ficheiro será nomeado como `cone.3d`.

Cada grupo de três linhas no ficheiro de saída corresponde a um triângulo, como exemplifica o excerto abaixo:

```
0.5 0 0
0 0 0.5
0.5 0 0.5
```

Listing 1: Triângulo - Excerto de Ficheiro .3d

2.1. Primitivas

Seguindo a lógica dos ficheiros .3d, aplicamos as seguintes primitivas:

- Plano
- Caixa
- Esfera
- Cone

Adicionalmente, decidimos desafiar-nos como grupo e implementar duas primitivas adicionais:

- Cilindro
- Torus

De seguida, exploramos em detalhe o processo de pensamento por de trás da geração de cada uma das primitivas.

2.1.1. Plano

Para uma correta geração do plano tem de ser conhecida a sua largura e o número de divisões a efetuar no mesmo. Para além disso, de forma a garantir que o plano está centrado na origem $(0, 0, 0)$, precisamos de dividir a sua largura por metade, o que denominamos *half*. Finalmente, para garantir que os pontos estão corretamente espaçados, dividimos a largura do plano pelo número de divisões existentes (*steps*).

Como sabemos que o plano está centrado na origem, deduzimos que para todos os pontos $y = 0$. Para x e z sabemos que estes estão dependentes da divisão em que se encontra, pelo que precisamos de iterar consoante o número da linha na vertical e horizontal i e j.

Assim, de modo a calcular os vértices de cada uma das divisões, aplicamos as seguintes fórmulas:

$$\text{x1} = \text{-half} + i * \text{steps}$$

$$\text{z1} = \text{-half} + j * \text{steps}$$

$$\text{x2} = \text{x1} + \text{steps}$$

$$z2 = z1 + steps$$

A figura que se segue ilustra a responsabilidade de cada uma destas variáveis:

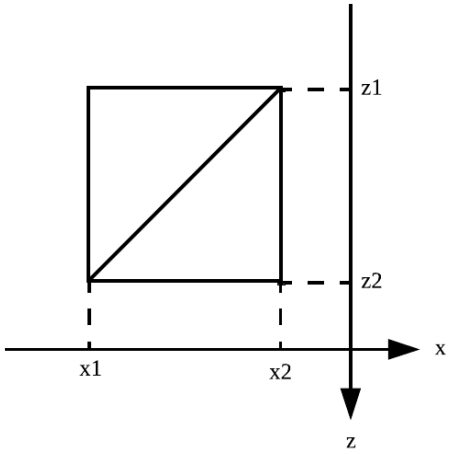

Figura 1: Ilustração de uma das divisões do plano

Com estes valores conseguimos então definir os quatros pontos necessários:

$$P1 = Point(x1,0,z1)$$

$$P2 = Point(x2,0,z1)$$

$$P3 = Point(x1,0,z2)$$

$$P4 = Point(x2,0,z2)$$

A figura abaixo demonstra como este cálculo progrede:

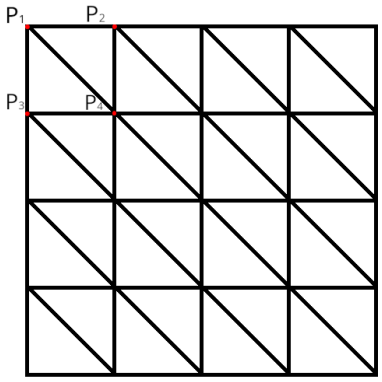

Figura 2: 1ª iteração do cálculo de pontos

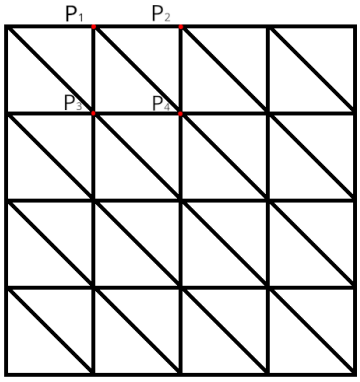

Figura 3: 2ª iteração do cálculo de pontos

Com o cálculo dos pontos estabelecido, precisamos apenas de garantir que a sua inserção no vetor é feita de forma ordeira, de maneira a conseguirmos desenhar os triângulos de forma correta.

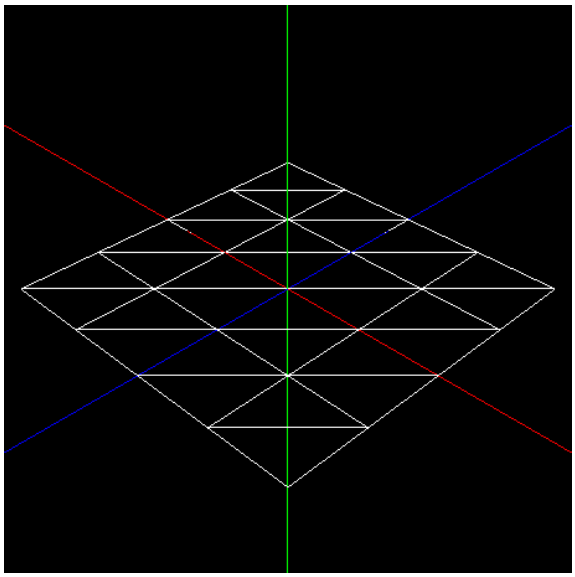

Figura 4: Plano com 2 de largura e 4 divisões

2.1.2. Caixa

De forma a gerar a caixa, decidimos optar por um método bastante similar ao do plano.

Tal como para o plano, a caixa está centrada na origem e é definida por uma largura e uma quantidade de divisões, calculamos metade da largura (*half*) e os espaços entre cada ponto (*steps*).

Para calcular as coordenadas x, y, z de cada vértice seguimos a mesma lógica do plano:

$$\text{v1} = \text{-half} + i * \text{steps}$$

$$\text{u1} = \text{-half} + j * \text{steps}$$

$$\text{v2} = \text{v1} + \text{steps}$$

$$\text{u2} = \text{u1} + \text{steps}$$

Tal como no plano, este valores serão usados para calcular os pontos. Porém o eixo em que estes valores serão usados dependerá da face em que estão. Por exemplo:

Face no plano x = half (Plano frontal da direita)	Face no plano y = half (Plano do topo)
$\text{P1} = \text{Point}(\text{half},\text{v1},\text{u1})$	$\text{P1} = \text{Point}(\text{v1},\text{half},\text{u1})$
$\text{P2} = \text{Point}(\text{half},\text{v2},\text{u1})$	$\text{P2} = \text{Point}(\text{v2},\ \text{half},\text{u1})$
$\text{P3} = \text{Point}(\text{half},\text{v1},\text{u2})$	$\text{P3} = \text{Point}(\text{v1},\ \text{half},\text{u2})$
$\text{P4} = \text{Point}(\text{half},\text{v2},\text{u2})$	$\text{P4} = \text{Point}(\text{v2},\text{half},\text{u2})$

Sendo assim, conseguimos calcular em cada iteração, uma divisão para cada face.

Por último, tal como para o plano, temos só que garantir a inserção destes pontos ordeiramente no vector.

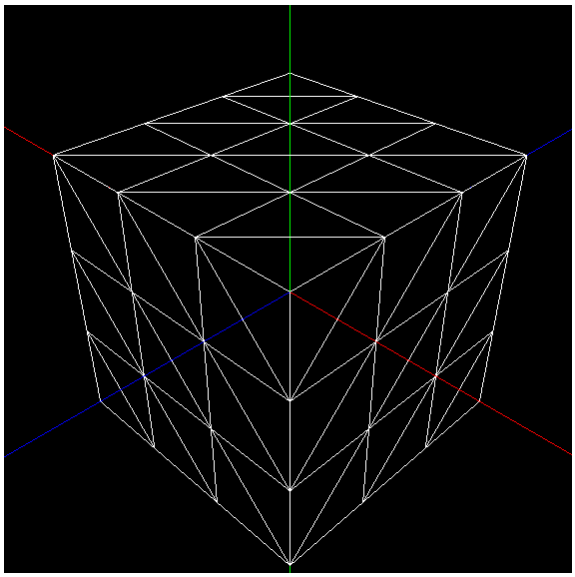

Figura 5: Cubo com 2 de largura e 3 divisões

2.1.3. Esfera

De maneira a construir a esfera, é necessário receber o raio, o número de *slices* e o número de *stacks*. Com os dados que recebemos, optamos por separar a sua construção em várias "fatias" verticais, neste caso as *slices*. Cada *slice* encontra-se dividida por um número de *stacks*, que correspondem ao número de divisões horizontais da esfera.

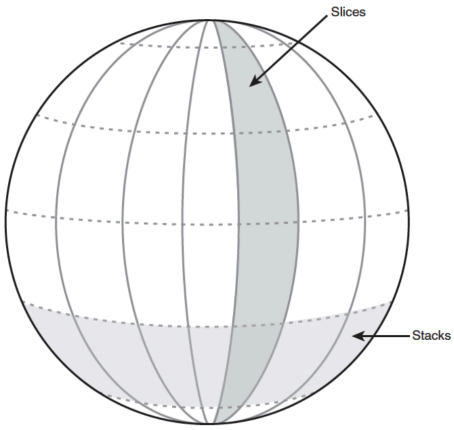

Figura 6: Ilustração da divisão da esfera

Após dividirmos a esfera conforme o número de *slices* e *stacks*, precisamos de calcular as coordenadas dos triângulos a construir de maneira fácil e repetitiva. Tendo em conta a forma final pretendida e a facilidade de calculo permitida, avançamos com o cálculo dos pontos através de coordenadas esféricas, que são convertidas no final para coordenadas cartesianas.

Para cada secção da esfera, calculamos um ângulo vertical φ e um ângulo horizontal θ que nos permitem establecer a base das coordenadas esféricas.

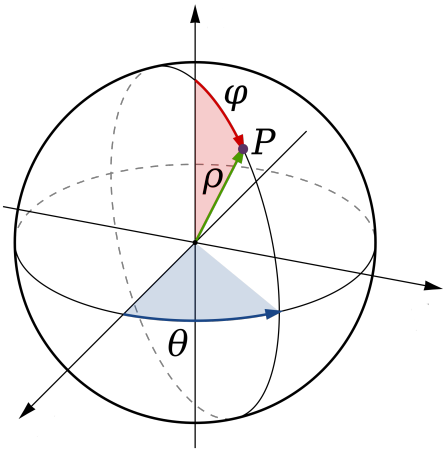

Figura 7: Ilustração das coordenadas esféricas

Onde φ e θ se obtêm da seguinte forma:

$$\varphi = \frac{\text{Slice\_atual} * \pi}{\text{Slices\_totais}}$$

$$\theta = \frac{\text{Stack\_atual} * 2\pi}{\text{Stacks\_totais}}$$

Podemos depois aplicar as seguintes fórmulas para converter as coordenadas esféricas para coordenadas cartesianas:

$$x = r * \sin(\theta) * \cos(\varphi)$$

$$y = r * \sin(\theta) * \sin(\varphi)$$

$$z = r * \cos(\theta)$$

Com o uso destas formulas é nos possível calcular a posição da cada vértice a utilizar na construção da figura. De maneira a facilitar os cálculos, para cada secção composta pelo *side* e pela *stack* atual, são calculados quatro pontos (dois na slice atual e dois na próxima) que permitem desenhar dois triangulos.

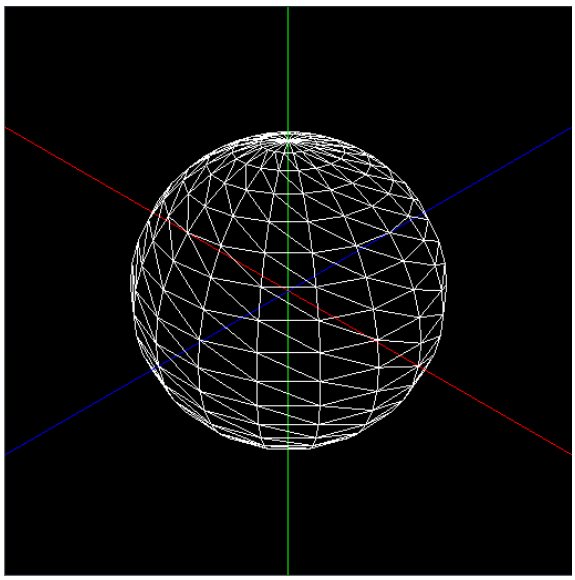

Figura 8: Ilustração de uma esfera com e de raio, 20 slides e 20 stacks

2.1.4. Cone

De forma a gerar o cone, são necessários os seguintes parâmetros: um raio, uma altura, o número de *slices* e o número *stacks*, onde as *slices* correspondem às divisões equivalentes do cone e as *stacks* às divisões equivalentes da altura do cone.

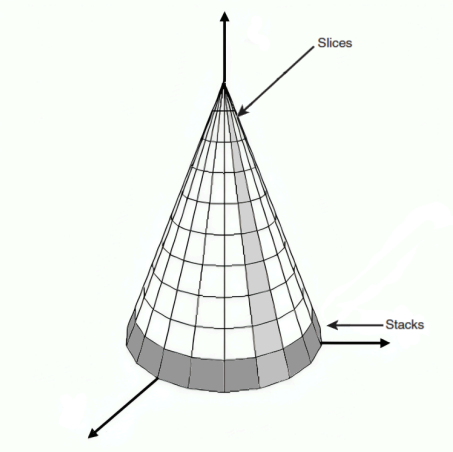

Figura 9: Ilustração de uma slice e uma stack de um cone

Para cada *stack*, é necessário calcular a sua altura e ângulo, que se obtêm da seguinte forma:

$$\text{Altura de cada Stack } (h) = \frac{\text{Altura total}}{\text{Nº de Stacks}}$$

$$\text{Ângulo para cada Stack} = -\left(\frac{\text{Raio do cone}}{\text{Nº de Stacks}}\right)$$

Para cada *slice*, é necessário calcular o seu ângulo, que se obtém da seguinte forma:

$$\text{Ângulo de cada Slice } (\alpha) = \frac{2\pi}{\text{Nº de Slices}}$$

A altura total do cone é recebida como argumento e prova-se fácil deduzir a altura dos pontos em cada iteração. Seja h o valor da altura total divido pelo número de *stacks* existentes, mencionado anteriormente, e y o valor da altura da *stack* atual em relação à origem:

$$y = n * h$$

Onde n é o número da *stack* atual.

Já o x e o z são calculados da seguinte forma:

$$x = \text{Raio da Stack} * \sin(\alpha)$$

$$z = \text{Raio da Stack} * \cos(\alpha)$$

Este processo é repetido iterativamente, sendo gerados, para cada *slice*, os triângulos de cada *stack*, pela ordem correta. Na implementação, isto traduz-se na execução de dois ciclos aninhados:

```
for each slice:
  for each stack:
    generateTriangle1()
    generateTriangle2()
```

Listing 2: Pseudo-código de ilustração ao processo de geração do cone.

Chegamos assim ao seguinte resultado renderizado:

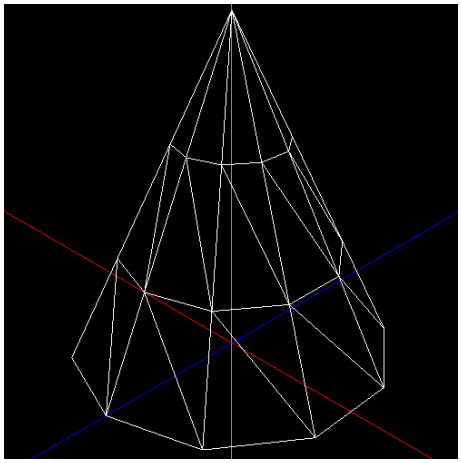

Figura 10: Ilustração de um cone com raio 1, altura 2, 10 slices e 3 stacks

2.1.5. Cilindro

A nossa abordagem na construção do cilindro passou por desenhar cada uma das suas *slices* de forma completa, ou seja, o triângulo da base, os dois triângulos que compõe a face e o triângulo do topo. Isto permite que o desenho da primitiva seja efetuado num só ciclo, em vez de três ciclos (base, superfície lateral e topo). A imagem que se segue ilustra este processo, destacando os triângulos gerados a cada iteração do ciclo.

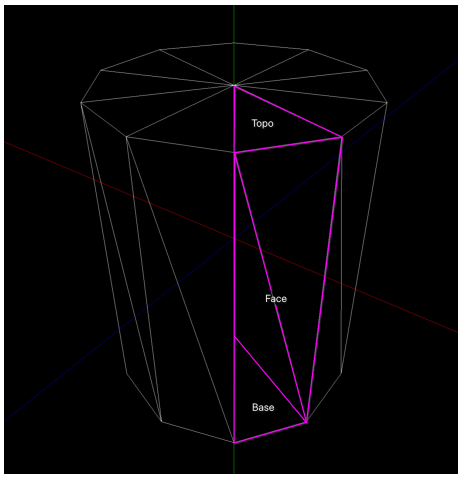

Figura 11: Ilustração do processo de desenho do cilindro

O cálculo de cada vértice é efetuado com a ajuda de coordenadas cilíndricas, cujo funcionamento se ilustra na figura seguinte:

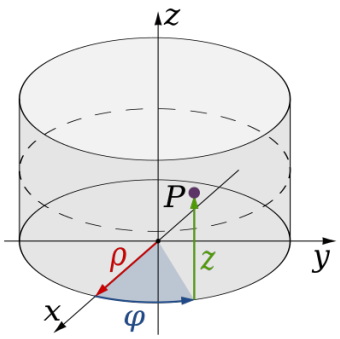

Figura 12: Ilustração do sistema de coordenadas cilíndricas

De acordo com o número de 'fatias' definido para o desenho da primitiva, é calculado o tamanho que cada fatia deverá tomar, *i.e* um ângulo α obtido da seguinte forma, onde n é o número de fatias:

$$\alpha = \frac{2\pi}{n}$$

A cada iteração do ciclo, o valor de φ é incrementado, tomando um valor múltiplo de α, de modo a calcular as coordenadas cilíndricas dos vértices da fatia atual. Estas coordenadas são posteriormente convertidas para coordenadas cartesianas através da seguinte fórmula, onde, para um ponto de coordenadas cilíndricas $P(\varphi, \rho, z)$, o seu equivalente no sistema de coordenadas cartesiano, $P(x, y, z)$, é tal que:

$$x = \rho * \cos(\varphi) \wedge y = \rho * \sin(\varphi) \wedge z = z$$

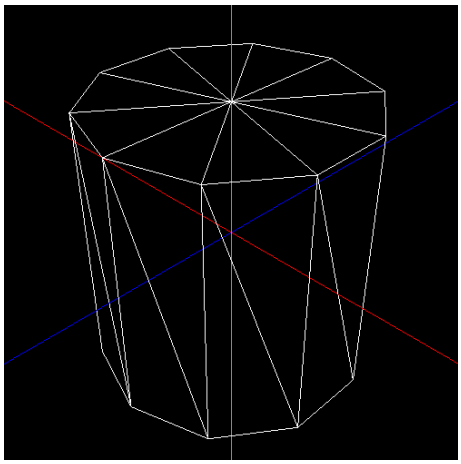

Figura 13: Ilustração de um cone cilindro de 1 de raio, 2 de altura e 10 slices

2.1.6. Torus

Como o torus se apresenta como uma primitiva extra, tivemos maior poder sobre o tipo de valores que deveremos receber para a sua construção. Optamos então por receber como argumentos o raio maior, o raio menor, número de *sides* e número de *rings*. Na figura abaixo podemos verificar que o valor R corresponde ao raio maior e o valor r corresponde ao raio pequeno.

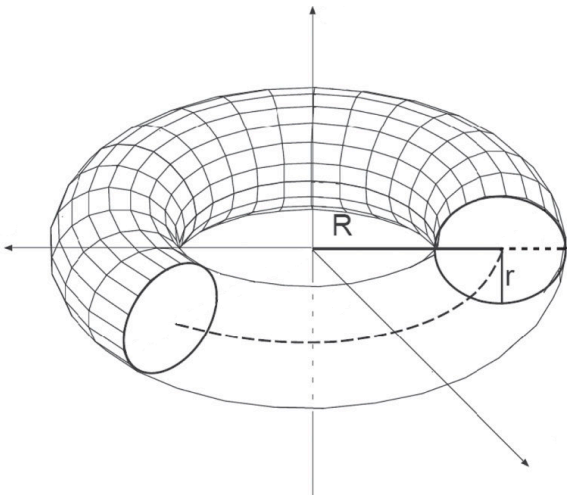

Figura 14: Ilustração da divisão de um torus

Os *rings* e *sides*, como os nomes indicam, correspondem aos anéis e lados que compõem o torus, correspondendo estes respetivamente às suas divisões verticais e divisões horizontais.

De maneira similar à esfera, desenhamos o torus um *ring* de cada vez, calculando a posição dos pontos com as seguintes expressões:

$$\varphi = \frac{\text{Side\_atual} * 2\pi}{\text{Sides\_totais}}$$

$$\theta = \frac{\text{Ring\_atual} * 2\pi}{\text{Rings\_totais}}$$

$$x = (R + r * \cos(\varphi)) * \cos(\theta)$$

$$y = r * \sin(\theta)$$

$$z = (R + r * \cos(\varphi)) * \sin(\theta))$$

Com o uso destas fórmulas é nos possível calcular a posição da cada vértice a utilizar na construção da figura. De maneira a facilitar os cálculos, para cada secção composta pelo *side* e pelo *ring* atual, são calculados quatro pontos (dois na *slice* atual e dois na próxima) que permitem desenhar dois triângulos.

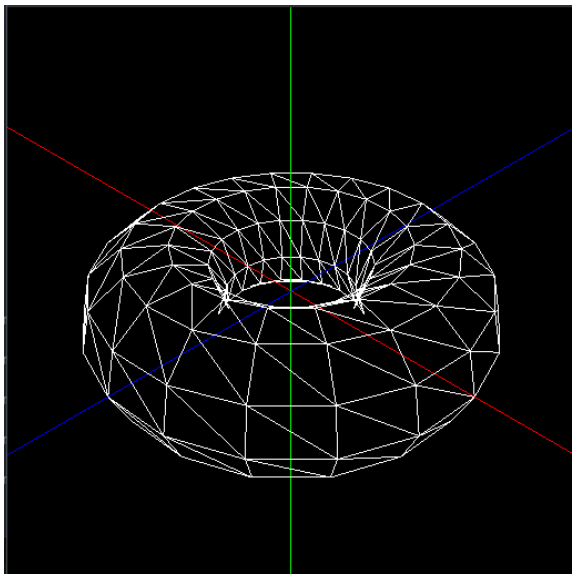

Figura 15: Ilustração de um torus com raio maior de 2, raio menor de 1, 20 rings e 10 sides

3. Engine

Para viabilizar a visualização das primitivas, foi concebido um **engine** 3D. Este **engine** consegue ler e intrepretar tanto ficheiros `.3d`, como ficheiros `.obj` e também cenas (ficheiros `.xml`).

Uma das responsabilidades do engine é ler e interpretar um ficheiro `.xml` de estrutura bem definida que contém informações cruciais, nomeadamente:

- Tamanho da janela a ser criada (`window`);
- Características da câmera (`camera`);
- Nome dos ficheiros `.3d` gerados pelo *generator* ou dos ficheiros `.obj` (`models`)

De forma não menos importante, deve ler e interpretar o conjunto de ficheiros `.3d` ou `.obj` lidos, por forma a renderizar as primitivas que estes definem através dos seus pontos.

As informações da câmera permitem que esta seja posicionada e configurada da forma definida. No entanto, para além disso, o *engine* implementa também a noção de movimento através da mesma, permitindo que o utilizador vizualize a cena de diferentes posições e ângulos, como por exemplo o movimento orbital em torno de um ponto fixo (normalmente o centro da primitiva renderizada).

3.1. Renderização da Cena

De maneira a conseguirmos renderizar as cenas foi necessário a criação de dois *parsers*, sendo um destes o leitor de XML e outro o leitor dos modelos.

3.1.1. Leitura e Interpretação do XML

A leitura e interpretação dos ficheiros XML é suportada pela biblioteca **RapidXML**. Com a ajuda desta biblioteca, os campos do ficheiro são explorados de acordo com a estrutura definida e os seus valores são extraídos para uma estrutura de dados dedicada, de modo a facilitar o acesso à informação pela aplicação. Esta estrutura de dados é suportada por um conjunto de classes, criadas para representar a informação de forma encapsulada e modular.

3.1.1.1. `Configuration`

A classe **`Configuration`** inclui toda a informação relativa à configuração descrita por cada ficheiro XML, onde a informação da janela é representada por um objeto do tipo `Window`, a informação da câmera por um objeto do tipo `Camera` e o nome dos modelos das primitivas a renderizar por um vetor.

3.1.1.2. `Window`

A classe **`Window`** contém a largura e comprimento (`width` e `height`) da janela a criar pelo engine.

3.1.1.3. `Camera`

A classe **`Camera`** contém três objetos do tipo `Point` que definem a sua posição, direção e orientação (`position`, `lookAt` e `up`, respetivamente). Para além disso contém também informação acerca do seu campo de visão e os limites superior e inferior do mesmo (`fov`, `near` e `far`, respetivamente).

3.1.2. Renderização dos Modelos

De maneira a conseguirmos renderizar os ficheiros .3d e .obj, guardados na **`Configuration`**, temos a função `parseOBJfile()` e a função `parse3Dfile()`, que ambas retornam listas de pontos. Esta lista de pontos irá ser guardada em memória e assim será chamada uma função que desenha cada um dos triângulos.

No caso do `parse3Dfile()`, devido à maneira como o ficheiro está estruturado, irá ler linha a linha, sendo cada linha um ponto e cada conjunto consecutivo de 3 linhas um triângulo.

Enquanto isso o `parseOBJfile()` funciona de uma forma diferente devido à maneira como os ficheiros `.obj` estão estruturados. Os ficheiros `.obj` geralmente encontram-se divididos em quatro tipos de dados diferentes, os vértices, as texturas, as normais dos vértices e as faces dos objetos. A parte inicial deste tipo de ficheiro contém as coordenadas dos vértices que compõe o objeto. Todas as linhas que possuem as coordenadas dos vértices são inicializadas por um identificador `v`. Pela mesma lógica, os restantes tipos de dados possuem identificadores prórios, mas, para a fase atual, só nos interessa os vértices e as faces, identificadas por um `f`. A lista de vértices originalmente fornecida não se encontra na ordem correta para desenhar a figura desejada, necessitando então que seja atribuída uma ordem específica. Esta ordem corresponde à ordem das faces, que indicam os índices dos vértices que compõem cada polígono do objeto, que, neste projeto, são triângulos.

Após diversas tentativas com diferentes ficheiros `.obj`, reparamos que o método utilizado para *parsing* só permite ficheiros com *triangular mesh* , o que provoca erros de desenho em ficheiros que possuem polígonos com mais, ou menos, do que 3 vértices.

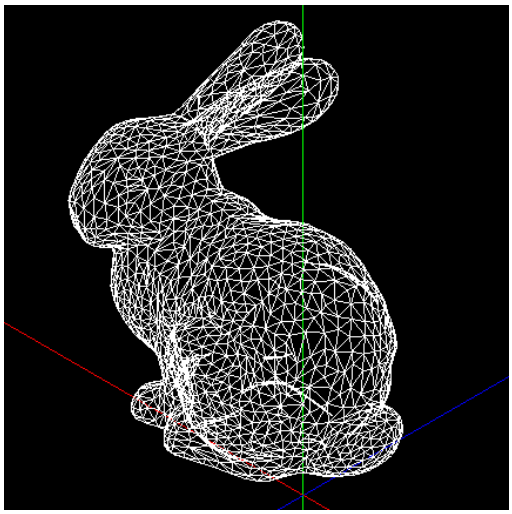

Figura 16: Ficheiro bunny.obj

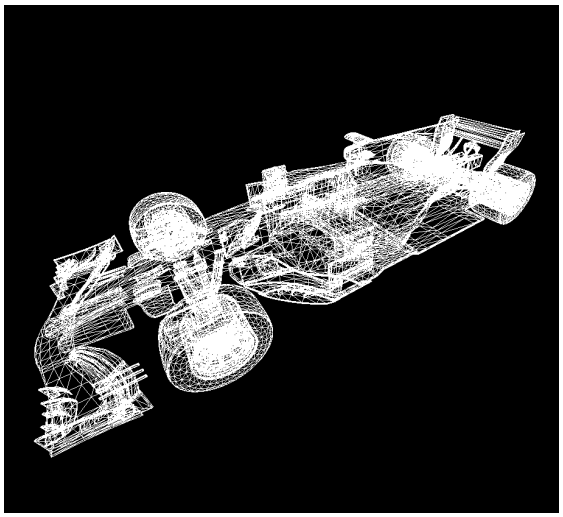

Figura 17: Ficheiro ferrari.obj - Carro de F1 da Ferrari

Obtendo então as listas de pontos, o nosso *engine* enviará as mesmas para uma função `drawTriangles()` que, como o nome indica, irá desenhar cada um dos triângulos. Esta, para cada ponto, chama a função `glVertex3f()` que implica uma chamada à GPU (Graphical Processing Unit). No futuro, planeamos substituir essa implementação por uma que envie diretamente os valores de todos os pontos para a GPU.

3.2. Exploração da Cena

De maneira a conseguir tornar a observação dos modelos mais interessante, decidimos implementar algumas *features*.

3.2.1. Movimento Orbital

Para conseguirmos ver os modelos de diferentes perspetivas implementamos uma câmera orbital. Isto é, uma câmera que move em torno de um ponto (sendo este o ponto definido pelo *lookAt*). Desta maneira conseguimos rodar a projeção e observar melhor o modelo.

3.2.2. Zoom

Para além disso, para situações como modelos bastante pequenos ou modelos maiores do que o esperado, conseguimos também fazer *zoom in* tal como *zoom out* no modelo. Para isso utilizamos as teclas "I" e "O".

3.2.3. Outros

Por último, de maneira a conseguirmos observar melhor a cena, temos ainda mais duas possibilidades. Podemos simplesmente desativar a projeção dos eixos, utilizando a tecla "A", e ainda conseguimos também repor a posição da câmera, voltando esta para a posição inicial, utilizando a tecla "R".

4. Utilitários

4.1. Bibliotecas

Para garantir o funcionamento adequado da aplicação, foi essencial integrar uma biblioteca específica. No caso, optamos pela **RapidXML**, uma biblioteca especializada na manipulação de arquivos XML, foi implementada para a leitura dos arquivos que definem as características da câmera e os modelos a renderizar, fornecendo uma solução eficiente e robusta para essa tarefa específica.

4.2. Scripts

De forma a facilitar o processo de desenvolvimento, foram criados dois scripts.

O primeiro é o script de **build**, que é responsável por compilar toda a aplicação, incluindo o *generator* e o *engine*.

O segundo é o script de **format**, que formata todos os arquivos `.cpp` e `.hpp` para garantir a consistência do código-fonte.

Ambos os scripts podem ser encontrados na pasta *scripts* do projeto.

4.3. Classes Comuns

Para tornar o projeto mais genérico e modular, foi criada a classe **`Point`** que representa cada ponto no plano tridimensional (x, y, z). Cada primitiva é então representada por um vetor de objetos **`Point`**.

5. Conclusão e Trabalho Futuro

A implementação deste programa proporcionou uma oportunidade de familiarização com a linguagem de programação C++ e com OpenGL, juntamente com as suas ferramentas para a cadeira de Computação Gráfica. Esta experiência estabeleceu as bases para desenvolver as restantes fases do projeto.

No entanto, pretendemos aprimorar diversos pontos do projeto. Para começar, planeamos alterar a maneira como guardamos os nossos ficheiros `.3d`, de modo a conseguirmos utilizar conceitos que consideramos essenciais como VBOs e indexação. Ainda mais, pretendemos desenvolver uma pequena GUI (Graphical User Interface) de maneira a auxiliar a utilização do *engine*. Por último, consideramos que será benéfico para o projeto adicionar conceitos como *frustsum culling*, uma técnica que nos permite otimizar o nosso *engine*, de forma a renderizar apenas o que está dentro do campo de visão da câmera, reduzindo assim a carga de processamento. Para além disso, o uso de VBOs e indexação mencionados anteriormente são medidas adicionais que também minimizam o número de chamadas à GPU.