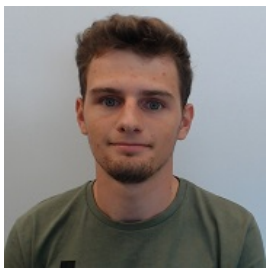
Universidade do Minho
Licenciatura em Engenharia Informática

Computação Gráfica

Trabalho Prático - Fase 3

Grupo 33


Bohdan Malanka
a93300


Diogo Rebelo
a93278


Henrique Alvelos
a93316


Lídia Sousa
a93205

8 de junho de 2022

Conteúdo

Lista de Figuras

1 Introdução

O presente documento é alusivo à **terceira** fase do projeto prático desenvolvido com recurso à linguagem de programação C++, no âmbito da Unidade Curricular de Computação Gráfica que integra a Licenciatura em Engenharia Informática da Universidade do Minho. Este projeto encontra-se dividido em quatro fases de trabalho, cada uma com uma data de entrega específica. Esta divisão em fases, pretende fomentar uma simplificação e organização do trabalho, contribuindo para a sua melhor compreensão.

Pretende-se, assim, que o relatório sirva de suporte ao trabalho realizado para esta fase, mais propriamente, dando uma explicação e elucidando o conjunto de decisões tomadas ao longo da construção de todo o código fonte e descrevendo a estratégia utilizada para a concretização dos principais objetivos propostos, que surgem a seguir:

- Compreender a utilização do *OpenGL*, recorrendo à biblioteca GLUT, para a construção de modelos 3D;
- Aprofundar temas alusivos à produção destes modelos 3D, nomeadamente, em relação a transformações geométricas, curvas, superfícies, iluminação, texturas e modo de construção geométrico básico;
- Relacionar todo o conceito de construir modelos 3D com o auxílio da criação de ficheiros que guardam informação relevante nesse âmbito;
- Relacionar aspetos mais teóricos com a sua aplicação a nível mais prático.

Naturalmente, é indispensável que o conjunto de objetivos supracitados seja concretizado com sucesso e, para isso, o formato do relatório está organizado de acordo com uma descrição do problema inicial, seguindo-se o conjunto de aspetos relevantes em relação ao **Gerador** e chegando aos aspetos primordiais sobre o **Motor**. O grupo decidiu incluir também uma secção direccionada para a descrição das funcionalidades adicionais.

2 Descrição do Problema

Nesta terceira fase do projeto o objetivo é desenvolver novas funcionalidades no **Gerador**, nomeadamente criar um novo modelo baseado nas superfícies de Bezier. Ou seja, receber um nome de um ficheiro que possui vários pontos de controlo. O resultado final será um ficheiro com vários triângulos que representam a superfície.
Quanto ao **Motor**, as rotações e translações deixarão de poder ser apenas estáticas. Na rotação o atributo "Angle"poderá ser trocado por "Time", representando o tempo em segundos a fazer uma rotação completa. O campo Translação pode conter uma lista de pontos que representam uma curva Catmunll-Rom e além disso pode ter um campo "Align"para especificar que o objeto necessita de estar alinhado com a curva.
Mais ainda, é necessário editar o ficheiro criado na fase anterior com o objetivo de desenhar um modelo do sistema solar dinâmico, incluindo um cometa com a trajetória definida usando uma curva Catmull-Rom.

3 Estrutura do projeto

Tal como na primeira fase a estrutura do projeto é mantida, isto é, as aplicações são divididas por diretorias de *Generator*, *Engine* e *Models*.

De um modo geral, a aplicação é construída pelas seguintes componentes:

- **Gerador:** contém o conjunto de funções e estruturas responsáveis por gerar o ficheiro com o conjunto de pontos de cada modelo, com a extensão `.3d`;
- **Motor:** contém o conjunto de funções e estruturas responsáveis por ler o ficheiro de configuração XML e representar graficamente cada modelo; Desta vez, para melhor organização do código, dividimos o código em vários ficheiros:
  - Funções relacionadas com o *parsing* dos ficheiros foram movidas para **parserXML.cpp**
  - Funções relacionadas com o cálculo da curva de Catmull-Rom foram definidas em **catmull-rom.cpp**
  - Estruturas foram alteradas para **dataStructs.h**
- **Models:** diretoria onde os ficheiros .3d e os ficheiros XML ficam guardados.

4 Gerador

No programa *Generator* foi necessário efetuar algumas alterações de modo a efetuar a leitura dos *patches de Bezier* e a sua transformação em pontos que formam triângulos.

Leitura dos Patches

Para a leitura do ficheiro .patch e para seguir com a estrutura do ficheiro começou-se por percorrer todas as linhas que representam cada patch, constituídos por 16 números inteiros relativos aos índices dos pontos de controlo. Desta forma, decidiu-se armazenar cada linha do ficheiro num vetor de inteiros e este num outro vetor aninhado para cada linha. Os pontos foram guardados num vetor de *Point* da seguinte forma:

```
vector<vector<int>> patches;
vector<Point> points;
```

Calculo dos pontos

Para formar os pontos constituintes do *teapot* começou-se por percorrer cada *patch* formando em cada iteração a matriz, *matrix*, que é calculada através da expressão abaixo.

$$matrix = \mathrm{M} \times \begin{pmatrix} \mathrm{P}_{00} & \mathrm{P}_{10} & \mathrm{P}_{20} & \mathrm{P}_{30} \\ \mathrm{P}_{01} & \mathrm{P}_{11} & \mathrm{P}_{21} & \mathrm{P}_{31} \\ \mathrm{P}_{02} & \mathrm{P}_{12} & \mathrm{P}_{22} & \mathrm{P}_{32} \\ \mathrm{P}_{03} & \mathrm{P}_{13} & \mathrm{P}_{23} & \mathrm{P}_{33} \end{pmatrix} \times \mathrm{M}^T$$

A matriz central será alterada em cada iteração à medida que se percorre o vetor *patches* sendo constituída pelas várias curvas de *bezier* que representa um ponto.

A matriz M é dada pela expressão abaixo mantendo-se constante ao longo de toda a formação de pontos. Como M é simétrica a sua transposta é igual a ela mesma.

$$\mathrm{M} = \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Para a multiplicação destas matrizes implementou-se a função *matrixMultiplication* que multiplica a matriz M pela matriz do *patch* e multiplica a matriz do *patch* pela transposta da matriz M, respetivamente.

De seguida, calculou-se cada ponto de *bezier, p(u,v)*, através da expressão seguinte:

$$p(u,v) = \begin{pmatrix} u^3 & u^2 & u & 1 \end{pmatrix} \times matrix \times \begin{pmatrix} v^3 \\ v^2 \\ v \\ 1 \end{pmatrix}, u, v \in [1, 0]$$

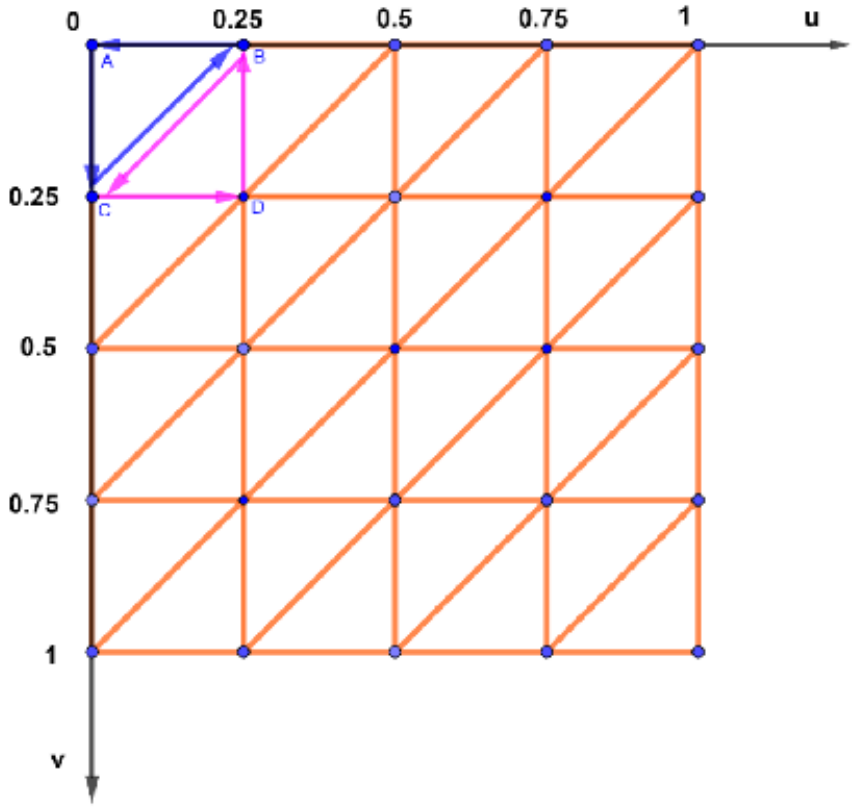
Para cada ponto formado, os parâmetros u e v vão aumentando conforme o nível de tesselação recebido, consoante o seguinte *step*:

$$step = \frac{1}{tesselagem}$$

Através da função *multMatrixVector* (que multiplica o vator u pela matriz *matrix* e o resultado multiplica pelo vator v) obtemos um ponto de *bezier* que será armazenado na matriz *grid*. Esta matriz é constituida por todos os pontos de *bezier* formados a partir de um *patch* como podemos ver de seguida:

P(0,0)	P(0.25,0)	P(0.5,0)	P(0.75,0)	P(1,0)
P(0,0.25)	P(0.25,0.25)	P(0.5,0.25)	P(0.75,0.25)	P(1,0.25)
P(0,0.5)	P(0.25,0.5)	P(0.5,0.5)	P(0.75,0.5)	P(1,0.5)
P(0,0.75)	P(0.25,0.75)	P(0.5,0.75)	P(0.75,0.75)	P(1,0.75)
P(0,1)	P(0.25,1)	P(0.5,1)	P(0.75,1)	P(1,1)

A matriz *grid*, no final das iterações u e v, contém todos os pontos de *bezier* que constituem um *patch* que formam uma gralha de quadrados. Para se conseguir representar o *teapot* por triângulos é necessário recorrer à triangulação da gralha, tal como está representado na figura seguinte.



Finalmente, respeitando a mão direita falta nos apenas escrever os pontos na ordem correta (A-C-B e C-D-B) para o ficheiro .3d que recebemos como parâmetro.

5 Motor

Nesta fase do projeto foi necessário atualizar o programa *Engine* com o principal objetivo de permitir translações e rotações com tempo e de desenhar as primitivas com recurso a VBOs. Para isso, foi necessário efetuar alterações tanto na leitura do ficheiro XML, como no desenho das primitivas. Nesse sentido, como o código cresceu muito decidimos dividir o código do ficheiro main.cpp em parseXML.cpp (responsável pela leitura e parse de XMl) e dataStructs.h contendo as estruturas de dados do programa.

A estrutura de dados teve que ser alterada para guardar os novos valores das rotações e translações (VBO?). Assim, a *struct Transform* ficou da seguinte forma:

```
struct Transform {
    string name;
    float x;
    float y;
    float z;
    float angle;
    float time = -1;
    bool align;
    vector<Point> points;
};
```

5.1 Leitura do ficheiro XML

De modo a suportar a extensão das transformações *rotate* e *translate*, foi necessário alternar a leitura do ficheiro XML. A função *parseInput*, que trata da análise deste documento e do povoamento das estruturas de dados, possui agora capacidade de ler e armazenar rotações e translações com o tempo.

5.2 VBOs

Para a implementação inspiramo-nos nas aulas práticas. Mesmo assim, não foi fácil o caminho até ter sucesso, isto porque não tínhamos muito bem a perceção de como funcionava esta realização. Mas, por fim, depois de ler o ficheiro .XML por inteiro, carregamos todos os vértices. Depois, ao processar os grupos, os vértices são desenhados.

5.3 Curvas de Catmull-Rom

Um dos novos requisitos desta fase era a implementação de animações.Desta forma, os corpos celestes presentes no nosso sistema solar, podem ser alvo de translações que emulam as suas órbitas em volta do sol, no caso dos planetas, ou volta dos planetas, como é ocaso das luas.

Para tal, foi necessária a implementação de uma das temáticas abordadas nas aulas: as curvas de Catmull-Rom. Estas vurvas são formadas por pelo menos 4 pontos(Po,P1,P2,P3), os quais manipulamos, em conjunto com uma iteração do valor t, através da equação abaixo apresentada:

$$p(t) = \begin{pmatrix} t^3 & t^2 & t & 1 \end{pmatrix} \times \begin{pmatrix} -0.5 & 1.5 & -1.5 & 0.5 \\ 1 & -2.5 & 2 & -0.5 \\ -0.5 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{pmatrix}$$

Desta forma podemos, por pontos fornecidos no XML do sistema solar, calcular a órbita de cada corpo celeste. De forma análoga se procede ao cálculo da posição de cada um dos corpos em cada frame, considerando-se o tempo passado em relação ao tempo total que pretendemos para a rotação à volta do Sol/Planeta.

5.4 Desenho das primitivas

Para desenhar as primitivas com a nova estrutura de dados foi criada a função *drawGroups*.

Esta função itera por cada grupo presente na estrutura principal *world.groups* e para cada um faz *push* da matriz das transformações através da função do *glut glPushMatrix*. Posteriormente

percorremos o *array* das transformações e executamos por ordem que aparece no ficheiro, atraves das funções *glTranslatef*, *glScalef* e *glRotatef*. Depois desenhamos a respetiva figura deste grupo com os pontos armazenados com ajuda da função *drawPrimitives*. Antes de fazer *pop* voltamos a chamar recursivamente a função (*drawGroups*) para executar as transformações dos subgrupos e só depois disso é que fazemos *glPopMatrix*. Deste modo, os subgrupos herdam as transformações do grupo pai.

5.5 Extras

Quanto aos extras, nesta fase implementamos a camera *First Person*, desenvolvida nas aulas práticas, para se poder navegar melhor pelas demos. No entanto, clicando com o botão direito do rato podemos alternar entre as duas cameras. Além disso, um menu com várias opções foi criado. Para o acessar, basta carregar no botão direito do rato. Este menu é constituído por 5 campos: **Camera**, **Linhas**, **Eixos**, **Ajuda** e **Sair**. Também se pode acessar ao menu **Ajuda** clicando na tecla **"h"**.

5.6 Modelo do sistema solar

Antes de desenvolver o modelo do sistema solar é preciso ter em consideração as seguintes propriedades:

- o *translate* de um grupo é somado ao do subgrupo;
- o valor do*scale* de um grupo é multiplicado ao valor do subgrupo;
- o valor do *rotate* do grupo é somado ao do subgrupo;

Para implementar a animação no sistema solar começamos por determinar os pontos das curvas de Catmull Rom que serão as órbitas dos planetas. O grupo decidiu que a órbita seria um círculo com o raio igual a distância do sol ao planeta. Assim, para auxiliar criamos uma script em *python* para gerar 16 pontos que formam a órbita.

```
import math
import sys

d =  float(sys.argv[2])

alpha =  (math.pi * 2) / 16
x = d
z = 0.0
y = 0.0

out = open("out.txt", "w")

if(sys.argv[1]=="planet"):
    out.write(f'<point x="{x}" y="0" z="{z}" />\n')

    i=15
    while i>0:
        angle = i * alpha
        x = float(d * math.cos(angle))
        z = float(d * math.sin(angle))
        out.write(f'<point x="{x}" y="0" z="{z}" />\n')
        i = i -1
elif(sys.argv[1]=="comet"):
    i=0
    a = 290.0
    b = 30.0
    h = 265.0
    k = 0.0
    while i<16:
```

```
        angle = i * alpha
        x = float(h + a*math.cos(angle))
        z = float(k + b*math.sin(angle))
        y = float(0.1068 * x - 3.3981)
        out.write(f'<point x="{x}" y="{y}" z="{z}" />\n')
        i = i + 1
```

Tendo as curvas das órbitas, falta agora calcular o tempo de translação de cada planeta. Começamos por analisar os valores reais e como não podemos aplicar uma só escala para representar a realidade, pois a discrepância é enorme entre os planetas telúricos e gasosos. Então, escolhemos qual seria o valor mínimo para percorrer uma órbita (Mercúrio=5s) e fomos a calcular uma diferença crescente para os próximos planetas:

```
M:                10
V: 10   + 1*1.5  =  11.5
T:11.5 + 2*2    =  15.5
//15.5 + 3*2.5  =  23
M:23   + 4*3    =  35

J:35   + 5*5    =  60
S:60 + 6*5.5    =  93
U:93 + 7*6      =  135
N:135  + 8*6.5  =  187
```

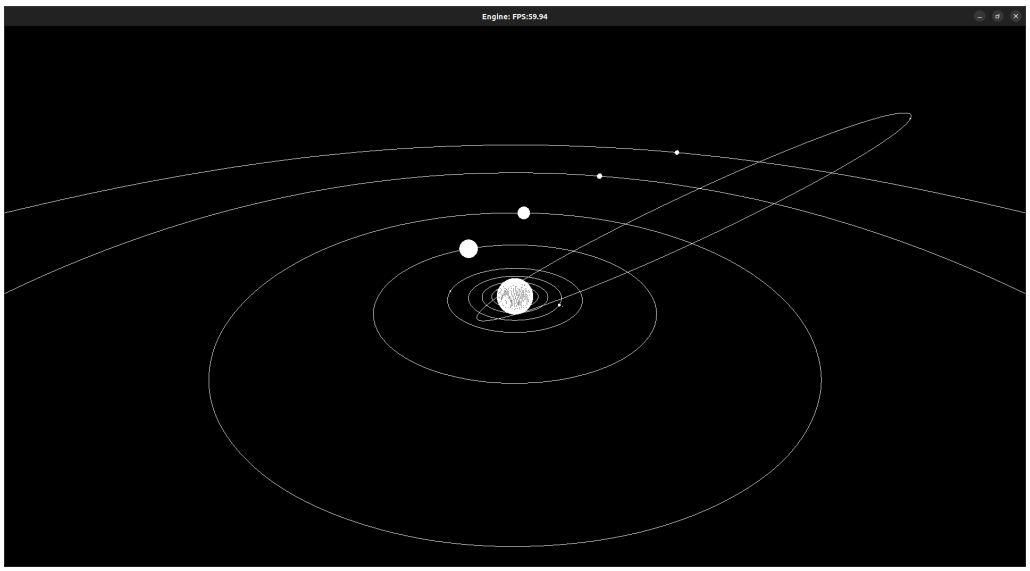
Quanto às rotações dos astros sobre si próprios, o método que utilizamos para calcular os tempos é semelhante ao das translações. Note-se que o Vénus e o Urano têm a rotação contrária dos outros planeta, pelo que o parâmetro *time* é negativo.

```
Tempo do Saturno completar uma rotação sobre si mesmo: 5s

J:            5
S: 5 + 1*1.5 = 6.5
N: 5 + 2*2   = 9
U: 5 + 2*2.5 = -10

T: 6 + 3*3   = 15
M: 6 + 3*3.5 = 16.5
Sol: 16.5 + 5*5.5 = 44
M: 44 + 6*6 = 80
V: 80 + 7*6.5 = -125,5
```

Quanto ao cometa, utilizamos a voluma de elipse para calcular a órbita como se pode observar no script supra-referido. A cordeada y, foi calculada utilizando uma função linear que passa nos pontos P1(500,50,0) e P2(-15,-5,0). Por fim, segue-se a imagem do novo sistema solar:


Engine: FPS:59.94

6 Testes

Para mostrar em funcionamento o trabalho exposto ao longo deste relatório realizamos os seguintes testes:

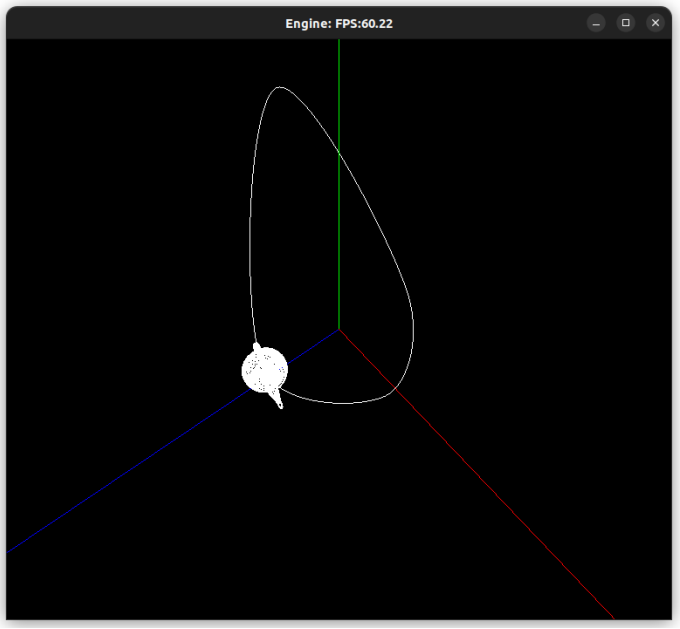


Figura 1: Ficheiro test_3_1.xml

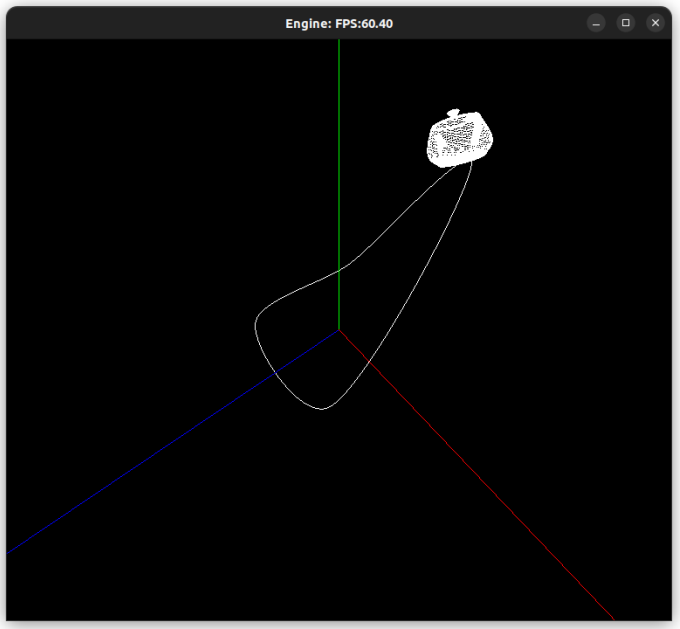


Figura 2: Ficheiro test_3_1.xml

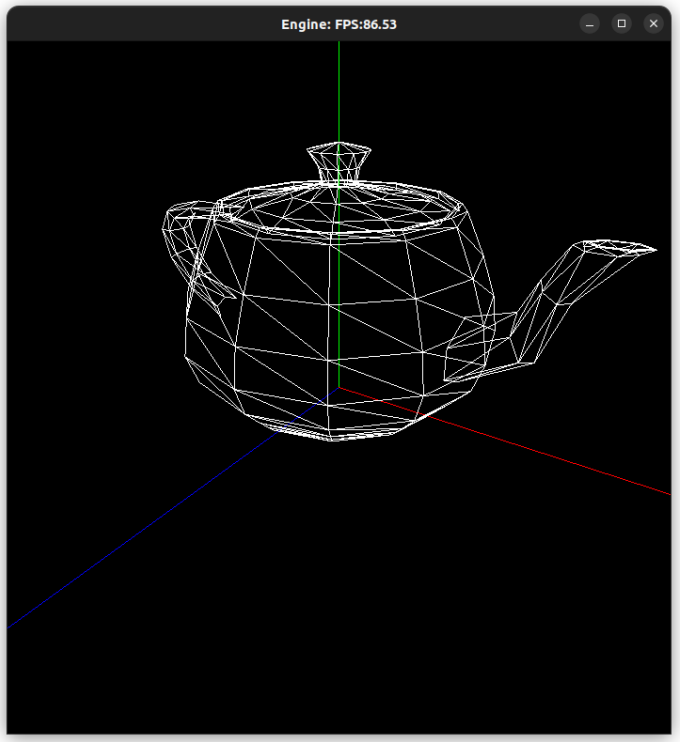


Figura 3: Ficheiro test_3_2.xml com tesselação 3

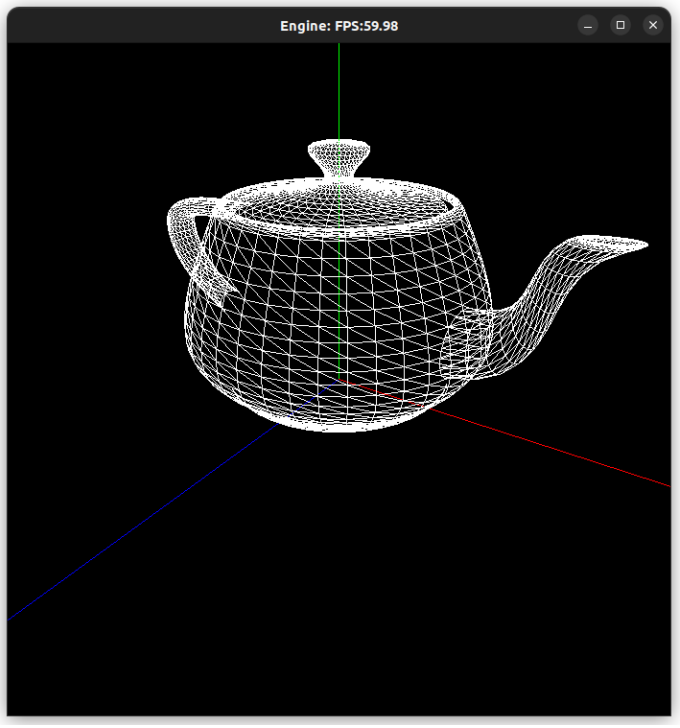


Figura 4: Ficheiro test_2_4.xml com tesselação 10

7 Conclusão

Da realização desta terceira fase, o grupo considera que a mesma foi bem conseguida, já que se realizaram todas as funcionalidades requisitadas pelo próprio enunciado.

No espetro positivo, consideramos conveniente destacar o correto funcionamento do nosso programa. Além disso, as estruturas implementadas estão em concordância com a estrutura do XML permitindo uma visualização mais clara daquilo que é armazenado.

Por outro lado, também existiram algumas dificuldades, tais como a familiarização com as funções do *tinyXML2*, a implementação de funções recursivas e a implementação dos VBO's. Apesar disso, as dificuldades foram ultrapassadas.

Para concluir, consideramos que houve um balanço positivo do trabalho realizado dado que as dificuldades sentidas foram superadas a ser cumpridos todos os requisitos.