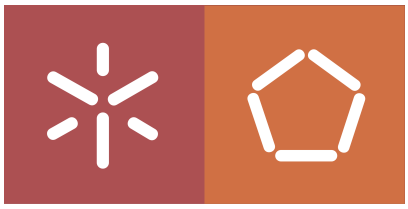
COMPUTAÇÃO GRÁFICA

Diogo Esteves
a104004

Rodrigo Fernandes
a104175

Diogo Barros
a100751

Maio 2025



Universidade do Minho
Escola de Engenharia

Conteúdo

1 Introdução

Este é um relatório relativo à quarta e última fase de desenvolvimento do motor gráfico 3D solicitado. Neste relatório iremos fazer um resumo do que foi desenvolvido até esta última fase, bem como iremos abordar as alterações relativas a esta fase, nomeadamente a adição de coordenadas normais e de textura às primitivas gráficas e a consequente utilização das técnicas de iluminação e textura.

Além do referido anteriormente, iremos concluir este relatório com uma análise crítica do projeto e possíveis melhorias ao mesmo.

2 *Generator*

O generator é a aplicação que gera os ficheiros '.3d' com os vértices de cada primitiva gráfica. Este tem em conta os argumentos passados pelo utilizador e, a partir deles, calcula precisamente os pontos das primitivas e escreve-os diretamente no ficheiro de output, com atenção à sua orientação, ou seja, a ordem pela qual os vértices são impressos está de acordo com a regra da mão direita.

Para que os vértices gerados cumpram a regra da mão direita, a ordem de desenho dos mesmos deve ter o sentido anti-horário. Assim, garantimos que todos os triângulos das primitivas estão voltados no corretamente segundo o eixo 'y', e por isso visiveis para a câmara.

A imagem abaixo exemplifica a ordem pela qual cada triângulo deve ser desenhado

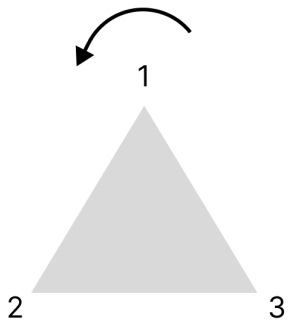


Figura 1: Exemplo de triângulo com face voltada para a câmara, segundo a regra da mão direita

Em cada ficheiro '.3d' gerado, é assinalada na primeira linha o nome da primitiva que foi gerada, para que a engine processe corretamente os dados facultados. Cada linha contém as coordenadas dos pontos no formato "X Y Z"e cada conjunto de 3 pontos (linhas) consecutivas corresponde a um triangulo que deve ser desenhado posteriormente pela *engine*. Ainda na linha que contém as coordenadas dos vértices de cada triângulo, estão presentes as coordenadas normais e de textura, sendo que o formato final de uma linha é o seguinte : 'X Y Z Nx Ny Nz Tx Ty'.

2.1 Primitivas

2.1.1 Plano

O plano é um quadrado paralelo ao plano XZ, centrado na origem e subdividido na direção X e Z. Para que seja possível construir o plano com recurso a triângulos, começamos por calcular dois parâmetros auxiliares, um *'step'* e um *'halfLength'*:

$$step = length/divisions \tag{1}$$

$$halfLength = length/2.0f \tag{2}$$

Após o cálculo dos parametros acima, podemos começar a iterar sobre os eixos relativos ao plano: no eixo X (variável i) e no eixo Z (variável j). A cada iteração, devemos cálcular as coordenadas dos pontos relativos ao triângulo que está a ser construido, segundo as seguintes fórmulas:

$$x1 = -halfLength + i * step \tag{3}$$

$$z1 = -halfLength + j * step \tag{4}$$

$$x2 = x1 + step \tag{5}$$

$$z2 = z1 + step \tag{6}$$

Como são necessários 3 pontos para a construção de um triangulo em openGL, vamos reciclar um dos pontos cálculados na iteração anterior. Assim, no ficheiro de output, um triângulo corresponde a uma sequência do tipo:

$x1$ 0.0 $z1$

$x1$ 0.0 $z2$

$x2$ 0.0 $z2$

$x1$ 0.0 $z1$

$x2$ 0.0 $z2$

$x2$ 0.0 $z1$

A sequência acima garante que a cada iteração, é construido um triângulo orientado em cada um dos eixos do plano e, assim, o output dos pontos gerados será um plano como:

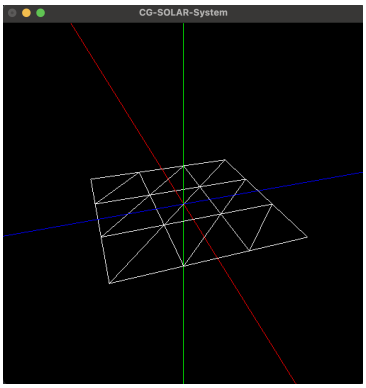


Figura 2: Plano

Para a adição das coordenadas normais, como se trata de um plano centrado na origem e paralelo aos eixos XZ, as coordenadas normais em cada um dos vértices dos triangulos constituintes do plano serão dadas por:

0.0 1.0 0.0

Por fim, para o cálculo das coordenadas de textura necessárias para mapear a área onde uma dada textura precisa de ser aplicada, utilizamos as seguintes fórmulas:

$$u1 = i/divisions \tag{7}$$

$$u2 = i + 1/divisions \tag{8}$$

$$v1 = j/divisions \tag{9}$$

$$v2 = j + 1/divisions \tag{10}$$

Assim, para a impressão de um quadrado formato por dois triângulos, temos que para cada linha a seguinte sequência em adição à sequência mencionada mais acima:

$u1\ v1$

$u1\ v2$

$u2\ v2$

$u1\ v1$

$u2\ v2$

$u2\ v1$

2.1.2 Caixa

A caixa (*box*) é um cubo centrado na origem do referêncial, e que varia de forma consoante parâmetros como a sua dimensão e o número de divisões por aresta.

Para começarmos a construção da *box*, com recurso a triângulos, calculamos dois parâmetros auxiliares:

$$halfDim = dimension/2.0f \tag{11}$$

$$step = dimension/divisions \tag{12}$$

Já com estes parâmetros auxiliares cálculados, podemos dar inicio à construção da *box*. Esta possui 6 faces, numeradas de 0 a 5, e cada face é subdividida em *divisions* × *divisions* quadrados. Para cada quadrado, são gerados dois triangulos, totalizando 6 vértices.

Para cada face, as coordenadas dos vértices são calculadas e escritas no ficheiro. A posição de cada vértice é definida em função das coordenadas (x1, y1), (x1, y2), (x2, y1), (x2, y2), onde:

- **x1 = -halfDim + j * step:** posição inicial do quadrado na direção x.
- **x2 = x1 + step:** posição final do quadrado na direção x.
- **y1 = -halfDim + i * step:** posição inicial do quadrado na direção y.
- **y2 = y1 + step:** posição final do quadrado na direção y.

Dependendo da face que está a ser processada, as coordenadas são ajustadas para corresponder à posição correta no espaço tridimensional.

Cada face é gerada utilizando um *switch(face)*, e os vértices são organizados para manter a orientação correta:

- **Face 0:** Frente (z = +halfDim)
- **Face 1:** Trás (z = -halfDim)
- **Face 2:** Esquerda (x = -halfDim)
- **Face 3:** Direita (x = +halfDim)
- **Face 4:** Topo (y = +halfDim)
- **Face 5:** Base (y = -halfDim)

Assim, após ler o ficheiro gerado, este deverá ser o output, por exemplo:

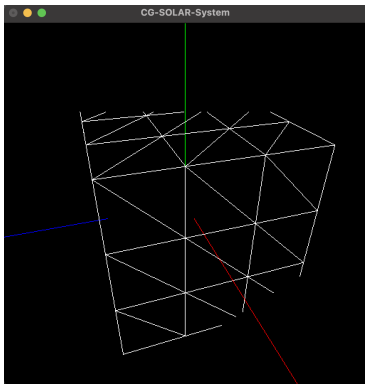


Figura 3: Box

À semelhança da primitiva anterior, nesta fase foram adicionadas as coordenadas normais e de textura a cada vértice dos triângulos da caixa. Para o cálculo das coordenadas normais e de textura, à semelhança do processo mencionado acima, utilizamos um *switch(face)*, para adicionar ao fim da linha as mesmas:

Face 0:

$$"x1\ y2\ halfDim\ tx1\ ty2"$$
$$"x1\ y1\ halfDim\ tx1\ ty1"$$
$$"x2\ y2\ halfDim\ tx2\ ty2"$$

$$"x2\ y2\ halfDim\ tx2\ ty2"$$
$$"x1\ y1\ halfDim\ tx1\ ty1"$$
$$"x2\ y1\ halfDim\ tx2\ ty1"$$

Face 1:

$$"x1\ y2\ -halfDim\ tx2\ ty2"$$
$$"x2\ y2\ -halfDim\ tx1\ ty2"$$
$$"x1\ y1\ -halfDim\ tx2\ ty1"$$

$$"x2\ y2\ -halfDim\ tx1\ ty2"$$
$$"x2\ y1\ -halfDim\ tx1\ ty1"$$
$$"x1\ y1\ -halfDim\ tx2\ ty1"$$

Face 2:

$$-halfDim\ y2\ x1\ tx2\ ty2"$$
$$-halfDim\ y1\ x1\ tx2\ ty1"$$
$$-halfDim\ y2\ x2\ tx1\ ty2"$$

$$-halfDim\ y2\ x2\ tx2\ ty2"$$
$$-halfDim\ y1\ x2\ tx2\ ty1"$$
$$-halfDim\ y1\ x1\ tx1\ ty1"$$

Face 3:

$$"halfDim\ y2\ x1\ tx1\ ty2"$$
$$"halfDim\ y2\ x2\ tx2\ ty2"$$
$$"halfDim\ y1\ x1\ tx1\ ty1"$$

$$"halfDim\ y2\ x2\ tx2\ ty2"$$
$$"halfDim\ y1\ x2\ tx2\ ty1"$$
$$"halfDim\ y1\ x1\ tx1\ ty1"$$

Face 4:

$$"x1\ halfDim\ y2\ tx1\ ty2"$$
$$"x2\ halfDim\ y2\ tx2\ ty2"$$
$$"x1\ halfDim\ y1\ tx1\ ty1"$$

$$"x2\ halfDim\ y2\ tx2\ ty2"$$
$$"x2\ halfDim\ y1\ tx2\ ty1"$$
$$"x1\ halfDim\ y1\ tx1\ ty1"$$

Face 5:

$$"x1\ -halfDim\ y2\ tx1\ ty2"$$
$$"x1\ -halfDim\ y1\ tx1\ ty1"$$
$$"x2\ -halfDim\ y2\ tx2\ ty2"$$

$$"x2\ -halfDim\ y2\ tx2\ ty2"$$
$$"x1\ -halfDim\ y1\ tx1\ ty1"$$
$$"x2\ -halfDim\ y1\ tx1\ ty1"$$

Onde tx1, tx2, ty1 e ty2 é dado por:

$$tx1 = j * textStep \tag{13}$$
$$tx2 = tx1 + textStep \tag{14}$$
$$ty1 = i * textStep \tag{15}$$
$$ty2 = ty1 + textStep \tag{16}$$

2.1.3 Cone

O desenho do cone requer parâmetros como:

- **radius:** Define o raio da base do cone.
- **height:** Define a altura total do cone.
- **slices:** Especifica quantas subdivisões circulares (fatias) compõem a base do cone.
- **stacks:** Especifica quantas camadas verticais o cone terá.

Além dos parametros listados acima, é necessário que a base do cone esteja no plano XZ. Para fazermos o desenho do cone com recurso a triangulos, começamos por calcular parâmetros que nos vão auxiliar na criação desta primitiva:

$$sliceStep = (2.0f * \pi)/slices \tag{17}$$
$$stackStep = height/stacks \tag{18}$$

Cálculados estes parâmetros auxilares, começamos por gerar uma base para o cone, contida no plano XZ. A base do cone é gerada conectando o centro (0,0,0) aos vértices ao longo da circunferência:

- **x = radius * sin(theta)** e **z = radius * cos(theta)** determinam as coordenadas circulares.
- Cada fatia gera um triangulo formado pelo centro e dois pontos consecutivos da circunferência.

De seguida, passamos pela criação das laterais do cone. Este é subdividido em *stacks* camadas, gerado-se assim vértices intermédios:

- **r1 = radius * (1.0f - j / stacks)**, **r2 = radius * (1.0f - (j + 1) / stacks)** reduzem o raio conforme a altura aumenta.
- Para cada fatia, dois triangulos são gerados conectando os vértices das camadas adjacentes.

Assim, um cone gerado pelo e processado pela engine, dependendo do ficheiro de configuração, porderá ser representado da seguinte forma:

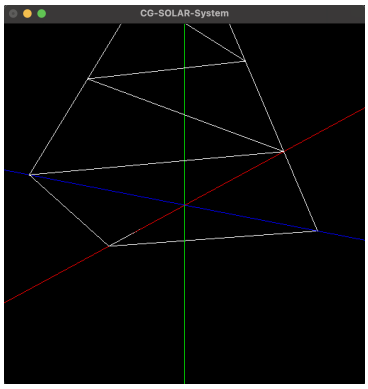


Figura 4: Cone

Para que a iluminação e a aplicação de texturas no cone funcionem corretamente, é necessário calcular normais e coordenadas de textura para cada vértice.

Na base do cone, as normais são constantes e apontam para baixo:

$$\vec{n}_{\text{base}} = (0.0, -1.0, 0.0) \tag{19}$$

As coordenadas de textura para os pontos da base são obtidas a partir da projeção circular no plano XZ, ajustadas para o intervalo [0,1]:

$$u = 0.5 + 0.5 \cdot \sin(\theta), \quad v = 0.5 + 0.5 \cdot \cos(\theta) \tag{20}$$

Nas laterais do cone, as normais devem ser perpendiculares à superfície inclinada, o que implica inclinação em relação ao eixo vertical. Para cada vértice lateral, a normal é definida como:

$$\vec{n} = (x, \frac{r}{h}, z), \quad \text{normalizada} \tag{21}$$

Onde x e z representam as coordenadas horizontais do vértice e $\frac{r}{h}$ representa a inclinação da superfície lateral (com r sendo o raio atual da camada e h a altura total). Esta normal é então normalizada para garantir comprimento unitário.

As coordenadas de textura para as laterais são definidas em função da fatia (i) e da camada (j):

$$u = \frac{i}{\text{slices}}, \quad v = \frac{j}{\text{stacks}} \tag{22}$$

Esta parametrização permite mapear a textura de forma contínua ao longo da superfície lateral do cone.

Com estas fórmulas, é possível gerar um modelo de cone com iluminação e texturização adequadas, possibilitando a sua correta visualização pela engine.

2.1.4 Esfera

Para conseguirmos fazer a criação de uma esfera utilizando triangulos, são necessários parâmetros tais como:

- **radius:** Define o raio da esfera.
- **slices:** Especifica o número de divisões ao longo da longitude.
- **stacks:** Especifica o número de divisões ao longo da latitude.

Além destes parâmetros, a esfera deve estar centrada na origem do referêncial. Assim, podemos começar a gerar as coordenadas da esfera,a partir da variação dos angulos theta (latitude) e phi (longitude):

- **theta**: varia de 0 a π, controlando a posição vertical.
- **phi:** varia de 0 a 2π, controlando a posição horizontal.

De seguida, passamos a cálcular as coordenadas dos vértices para cada divisão da esfera, da seguinte forma:

$$theta1 = i * \pi / stacks \tag{23}$$
$$theta2 = (i+1) * \pi / stacks \tag{24}$$
$$phi1 = j * 2\pi / stacks \tag{25}$$
$$phi1 = j * 2\pi / stacks \tag{26}$$

$$x = radius * sin(theta) * sin(phi) \tag{27}$$
$$y = radius * cos(theta) \tag{28}$$
$$z = radius * sin(theta) * cos(phi) \tag{29}$$

Por fim, é necessário escrever no ficheiro os triangulos relativos à esfera. Cada célula da malha da esfera é composta por dois triangulos, escritos no ficheiro na seguinte ordem:

$$x1\ y1\ z1$$
$$x4\ y4\ z4$$
$$x2\ y2\ z2$$

$$x1\ y1\ z1$$
$$x3\ y3\ z3$$
$$x4\ y4\ z4$$

Assim, um exemplo de uma esfera gerada pelo *generator* pode ser:

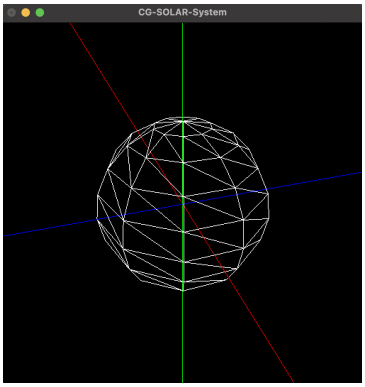


Figura 5: Esfera

Para garantir uma correta renderização da esfera com iluminação e texturas, é necessário calcular as normais e coordenadas de textura para cada vértice.

As **normais** da esfera são vetores que apontam radialmente a partir do centro da esfera. Como os vértices da esfera são posicionados com base em coordenadas esféricas, as normais podem ser obtidas diretamente a partir da posição normalizada de cada vértice:

$$\vec{n} = (\sin(\theta) \cdot \sin(\phi),\ \cos(\theta),\ \sin(\theta) \cdot \cos(\phi)) \tag{30}$$

Como o vetor normal coincide com a direção radial do ponto (e a esfera está centrada na origem), basta normalizar o vetor posição para obter a normal.

As **coordenadas de textura** são calculadas com base na variação angular dos vértices. Como a esfera é mapeada a partir de uma superfície bidimensional (imagem) sobre uma superfície tridimensional, utiliza-se uma parametrização esférica simples:

$$u = \frac{\phi}{2\pi}, \quad v = 1 - \frac{\theta}{\pi} \tag{31}$$

Estas fórmulas garantem que:

- u varia de 0 a 1 ao longo da longitude (phi).
- v varia de 0 a 1 ao longo da latitude (theta), com a inversão para garantir o alinhamento vertical correto da textura.

Desta forma, cada ponto da esfera tem associada uma normal unitária e um par de coordenadas de textura que permitem a sua correta iluminação e aplicação de textura ao ser processado pela *engine*.

2.1.5 Cilindro

O cilindro é um prisma, com duas bases circulares conectadas por uma superficie curva e centrado na origem do referêncial. As bases circulares são paralelas ao plano XZ e estão divididas em *´slices´*, para que seja possível fazer o seu desenho utilizando triângulos.

Assim, para o desenho das bases, é necessário dividir o ângulo da circunferência consoante o número de slices, segundo a seguinte fórmula:

$$\Delta = 2\pi/slices \tag{32}$$

Dado agora um Δ, é possível fazer o cálculo da posição dos pontos relativos à base da circunferência, utilizando sempre a origem do referêncial como ponto auxiliar, utilizando as seguintes fórmulas para o cálculo das coordenadas em x, y e z:

$$x1 = radius * sin(i * \Delta) \tag{33}$$

$$y1 = height/2 \tag{34}$$

$$z1 = radius * cos(i * \Delta) \tag{35}$$

Já calculado o primeiro ponto, como o openGL necessica de 3 pontos para formar o triangulo, é necessário cálcular o ponto seguinte e adicionar a origem:

$$x2 = radius * sin(i + 1 * \Delta) \tag{36}$$

$$y2 = (height/2) \tag{37}$$

$$z2 = radius * cos(i + 1 * \Delta) \tag{38}$$

$$ox = 0 \tag{39}$$

$$oy = +/- (height/2) \tag{40}$$

$$oz = 0 \tag{41}$$

Assim, no ficheiro, o output das 3 linhas correspondentes a um triangulo da base superior aparecem como:

$$x1\ y1\ z1$$

$$ox\ oy\ oz$$

$$x2\ y2\ z1$$

É necessário respeitar esta ordem, para que os triângulos cumpram a regra da mão direita e fiquem visiveis. Quanto à base do cilindro, o processo é idêntico, com a coordenada em y negativa. Em relação aos triângulos laterais, basta unir os triangulos das bases segundo a ordem correspondente, para que estes também fiquem visiveis.

Assim, o output do cilindro após a execução da engine será, por exemplo:

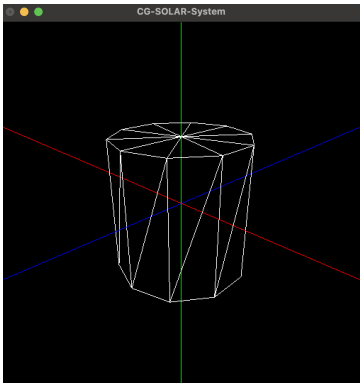


Figura 6: Cilindro

Como o cilindro não foi utilizado nem nos testes funcionais nem na nossa demo, optamos, para uma melhor gestão de tempo de desenvolvimento, por não adaptar o mesmo, pelo que este não contém coordenadas normais nem de textura.

2.1.6 Toro

A ideia do toro surgiu no desenvolvimento da demo do Sistema Solar, como uma *solução* para a representação gráfica dos anéis de Saturno.

O toro é uma geometria tridimensional em forma de anel, gerada pela rotação de um círculo ao longo de um caminho circular. O raio do círculo que é rotacionado é chamado de **raio menor**. Já o raio do caminho circular é denominado **raio maior**, que corresponde à distância do centro do toro até o centro do círculo que o define.

O toro foi gerado através de uma abordagem paramétrica, onde a sua superfície é definida pelos raios menor (r) e maior (R). Para criar a malha do toro, utilizou-se um sistema de coordenadas esféricas, dividindo a superfície em **fatias** (*slices*) e **pilhas** (*stacks*).

$$x(\theta, \phi) = (R + r \cdot \cos(\theta)) \cdot \cos(\phi) \tag{42}$$

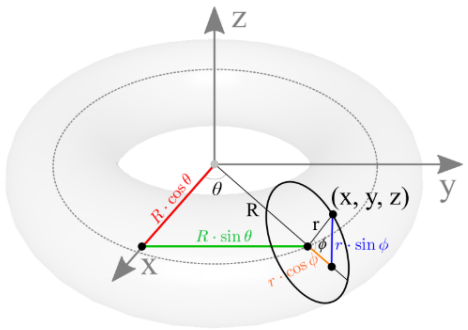


Figura 7: Exemplo de toro, sendo r o raio menor e R o raio menor

$$y(\theta, \phi) = (R + r \cdot \cos(\theta)) \cdot \sin(\phi) \tag{43}$$

$$z(\theta, \phi) = r \cdot \sin(\theta) \tag{44}$$

onde:

- θ é o ângulo de rotação ao redor do círculo menor (variando de 0 a 2π),
- ϕ é o ângulo de rotação ao redor do eixo central do toro (variando de 0 a 2π).

O intervalo $[0, 2\pi]$ é dividido em n fatias e m pilhas, com passos angulares dados por:

$$\Delta\theta = \frac{2\pi}{n}, \quad \Delta\phi = \frac{2\pi}{m}.$$

Para permitir uma iluminação realista e a aplicação de texturas no toro, é necessário calcular as normais e coordenadas de textura de cada vértice da malha.

As **normais** do toro são vetores perpendiculares à sua superfície e são obtidas a partir da orientação do círculo menor. Como o toro é gerado por rotação, a normal em cada ponto é dada pela direção que liga o centro do tubo (círculo menor) até o ponto na superfície. No caso da parametrização usada, as normais são dadas por:

$$\vec{n} = (\cos(\theta) \cdot \cos(\phi),\ \cos(\theta) \cdot \sin(\phi),\ \sin(\theta)) \tag{45}$$

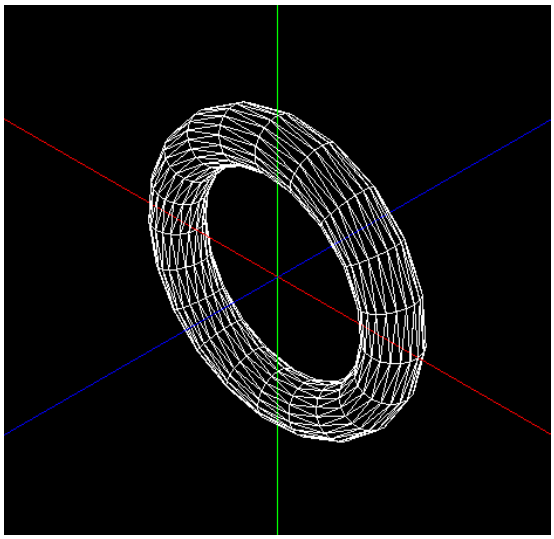


Figura 8: Toro de raio menor 2, raio maior 10, stacks 20, slices 20

Estas componentes representam a direção radial do círculo menor em torno da trajetória circular definida pelo raio maior. Como a geometria é regular, as normais já são unitárias e não necessitam ser normalizadas.

As **coordenadas de textura** mapeiam uma imagem bidimensional sobre a superfície do toro. Neste caso, os ângulos utilizados para parametrizar o toro são diretamente convertidos em coordenadas (u, v), normalizadas para o intervalo $[0, 1]$:

$$u = \frac{\phi}{2\pi}, \quad v = \frac{\theta}{2\pi} \tag{46}$$

- u varia ao longo do eixo do toro (ângulo ϕ),
- v varia ao longo do tubo (ângulo θ).

Assim, cada vértice do toro é composto pela sua posição tridimensional, vetor normal e coordenadas de textura, sendo estes os dados exportados para o ficheiro de modelo, utilizados posteriormente na renderização pela *engine*.

2.1.7 *Patch*

Nesta fase foi implementada a capacidade de gerar superfícies cúbicas de Bézier a partir de um ficheiro com patches de controlo e pontos. Cada patch define um conjunto de 16 índices que referenciam pontos de controlo num espaço tridimensional. O objetivo é, dada uma tesselação, produzir os triângulos que representam visualmente a superfície.

A geração das superfícies baseia-se no conceito de **superfícies de Bézier**, que são definidas por uma malha bidimensional de pontos de controlo $P_{i,j}$ com $i \in [0, m]$ e $j \in [0, n]$. A superfície de Bézier é uma função paramétrica de duas variáveis u e v, ambas no intervalo $[0, 1]$, dada pela seguinte equação:

$$S(u, v) = \sum_{i=0}^{m} \sum_{j=0}^{n} B_{m,i}(u) \cdot B_{n,j}(v) \cdot P_{i,j} \tag{47}$$

onde $B_{m,i}(u)$ e $B_{n,j}(v)$ são as funções base de Bézier, definidas por:

$$B_{n,i}(t) = \binom{n}{i} (1-t)^{n-i} t^i \tag{48}$$

Estas funções base, conhecidas como **funções de Bernstein**, determinam o peso de cada ponto de controlo na construção da superfície.

Cada patch é processado individualmente. Para cada par de parâmetros (u, v) definido pela tesselação, é calculado um ponto na superfície utilizando o produto tensorial das funções de Bernstein nos dois eixos. Isto permite interpolar a superfície a partir dos 16 pontos de controlo definidos no patch.

Para cada patch, a malha de pontos gerada com base na tesselação é usada para criar os triângulos. Cada célula da grelha é composta por dois triângulos definidos a partir dos quatro pontos do quadrado correspondente. Estes triângulos são então escritos no ficheiro de saída como coordenadas tridimensionais.

Esta abordagem permite representar superfícies suaves e contínuas com base num número reduzido de pontos de controlo, sendo extremamente útil para modelação de formas curvas em computação gráfica.

Seguem uns exemplos de um figuras geradas a partir de ficheiros patch:

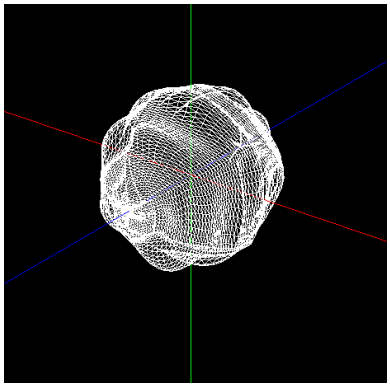


Figura 9: Cometa gerado com superfícies de Bezier

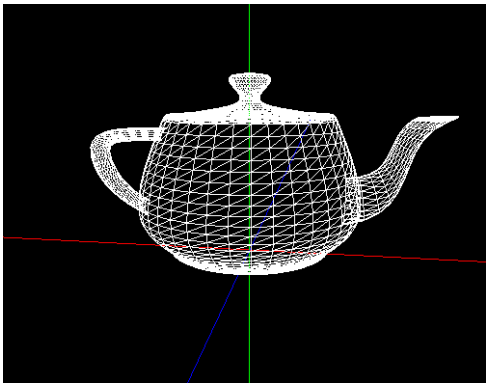


Figura 10: Teapot gerado com superfícies de Bezier

Para além da geração dos pontos da superfície, foram também calculadas as **normais** a cada triângulo gerado. Estas são essenciais para a correta iluminação dos modelos nas fases de renderização. A normal a um triângulo definido por três pontos P_1, P_2 e P_3 é obtida através do produto vetorial entre dois vetores do plano:

$$\vec{N} = \frac{(P_2 - P_1) \times (P_3 - P_1)}{\|(P_2 - P_1) \times (P_3 - P_1)\|} \tag{49}$$

Este vetor é depois normalizado para garantir que tem comprimento unitário. Cada triângulo possui a sua própria normal, e esta é atribuída a todos os seus vértices.

Adicionalmente, foram atribuídas **coordenadas de textura** a cada ponto da malha, permitindo futuramente aplicar texturas sobre as superfícies geradas. Como a superfície é parametrizada nos intervalos $u, v \in [0, 1]$, as coordenadas de textura associadas a cada ponto da grelha são diretamente dadas pelos valores dos parâmetros u e v usados na sua geração:

$$(u_{tex}, v_{tex}) = (u, v) \tag{50}$$

Desta forma, a textura mapeia-se de forma contínua e coerente sobre a superfície Bézier.

A combinação da interpolação paramétrica, do cálculo das normais e da atribuição das coordenadas de textura permite gerar superfícies suaves, com iluminação realista e capacidade de receber texturas, oferecendo uma representação visual rica e flexível a partir de um conjunto reduzido de pontos de controlo.

3 *Engine*

O principal objetivo da aplicação de *engine* é converter a informação gerada pelo *generator* e, com base em ficheiros de configuração XML, tornar essa informação visível ao utilizador. Para isso, utilizamos duas ferramentas para auxilio no desenvolvimento do programa:

- **Biblioteca GLut:** Biblioteca responsável pelo desenho dos modelos gerados anteriormente e criação de cenas 3D.
- **Biblioteca TinyXML:** Biblioteca auxiliar utilizada no processo de *parsing* dos ficheiros de configuração XML.
- **Biblioteca devIL:** Biblioteca auxiliar utilizada no processo de renderização de texturas a partir de imagens jpg.

3.1 Organização da Engine

Na programação com a biblioteca Glut, é necessário antes de entrarmos no ciclo principal, fazermos o *callback* das funções que serão utilizadas para as funções básicas na renderização da cena: função de desenho, *resize*, interação, etc. Como estas funções têm assinaturas *standard*, que devem ser respeitadas, foi definido um objeto 'global', que reflete a configuração presente nos ficheiros XML: **'Config'**.

Com a definição do objeto global no inicio da execução da engine garantimos que:

- **Existe persistência dos dados durante a renderização da cena:** O loop de renderização do OpenGL precisa de aceder aos modelos frame a frame para desenhá-los no ecrã. Como a função de desenho (draw()) é chamada repetidamente dentro do loop principal, os modelos precisam estar dispinveis num scope global.
- **Callbacks conseguem aceder a parametros:** Assim, mesmo respeitando a assinatura *standard* das funções do Glut, conseguimos aceder dentro do scope das mesmas a parametros necessários, como os presentes dentro dos ficheiros de configuração XML.

Assim, o objeto config contém um espelho em memória dos dados dos ficheiros XML, com os métodos necessários para o desenho da cena que está contida no mesmo.

3.2 *Parsing*

O processo de parsing é um processo crucial na execução do motor gráfico. Este processo não se limita apenas a ler o ficheiro XML e a criar estruturas em memória para conter os seus dados.

É durante o parsing que são aplicadas e guardadas numa matriz em memória as transformações estáticas presentes no ficheiro XML (tema que será abordado no ponto 3.5). Além disso, ainda durante o parsing, são criadas e enviadas para a memória gráfica as VBOs com os dados relativos às coordenadas dos vértices, coordenadas normais e de textura, bem como (se for o caso), são pré-processadas a iluminação e as texturas (temas a serem abordados nos pontos 3.4, 3.7 e 3.8 respetivamente).

3.3 Câmara Orbital

A câmara orbital foi implementada de forma a permitir ao utilizador observar a cena a partir de uma perspetiva esférica em torno de um ponto de interesse (habitualmente a origem do sistema de coordenadas). Esta abordagem é ideal para visualização de sistemas complexos, como o sistema solar, pois permite ao utilizador explorar livremente o espaço tridimensional mantendo um foco central constante.

A posição da câmara é calculada em coordenadas esféricas (r, θ, ϕ), onde:

- r representa a distância radial entre a câmara e o ponto de interesse;
- θ (theta) representa o ângulo de rotação em torno do eixo vertical (azimute);
- ϕ (phi) representa o ângulo de elevação (ângulo zenital) em relação ao plano horizontal.

Estas coordenadas esféricas são posteriormente convertidas para coordenadas cartesianas (x, y, z) utilizando as seguintes equações:

$$x = r \cdot \cos(\phi) \cdot \sin(\theta) \tag{51}$$

$$y = r \cdot \sin(\phi) \tag{52}$$

$$z = r \cdot \cos(\phi) \cdot \cos(\theta) \tag{53}$$

A câmara é, então, posicionada na cena com base nestas coordenadas e orientada para o ponto de interesse utilizando a função `gluLookAt`, que define a direção do olhar da câmara, o ponto de observação e o vetor *up*.

A interação com a câmara foi implementada utilizando o rato: ao pressionar e arrastar com o botão esquerdo, os valores dos ângulos θ e ϕ são atualizados com base no movimento horizontal e vertical do rato, respetivamente. Assim, o utilizador consegue "orbitar"em torno do centro da cena de forma fluida.

Para além disso, foi ainda implementada a funcionalidade de mostrar ou esconder os eixos da cena pressionando a tecla `T`, alternando o valor de uma variável booleana interna que controla esse comportamento visual.

Esta câmara proporciona uma navegação intuitiva e é particularmente útil para visualização exploratória, sendo uma abordagem comum em motores gráficos e aplicações de visualização 3D.

3.4 Otimizações de Desempenho

Durante o desenvolvimento do projeto foram implementadas as otimizações de desempenho proporcionadas pela implementação de VBOs (Vertex Buffer Objects), utilizadas para desenho da geometria do OpenGL.

Um VBO pode ser visto como um array alocado na memória da placa gráfica, ao invés de alocado na memória RAM (como implementado nas fases anteriores). Assim obtemos ganhos de desempenho significativos, uma vez que foi reduzida a comunicação entre o CPU e a GPU, uma vez que agora a informação relatica aos vértices encontra-se alocada localmente na GPU. Além disso, torna-se possível tirar partido do paralelismo da GPU, uma vez que a distribuição de carga é agora gerida pelo driver da GPU.

Para que fosse possível implementar VBOs para cada modelo de uma dada cena 3D, começamos por, durante o processo de parsing dos ficheiros '.3d', iniciar para cada modelo um array/vector em C e copiamos o mesmo para a GPU, associando ao mesmo um ID (para que mais tarde seja possível o desenho do mesmo) que fica também armazenado em memória para consulta posterior. Esta operação é realizada apenas no parsing, não se repetindo mais nenhuma vez durante a execução do programa, uma vez que a geometria do mesmo não é dinamica.

Durante o processo de desenho, para cada modelo basta passar o ID do VBO armazenado, para garantir que estamos a desenhar o modelo correto, e o tipo dos dados que o array que armazenamos posteriormente contém.

O processo mencionado acima é repetido para as coordenadas normais e de textura, pelo que por cada modelo que o XML contenha, são criados 3 VBOs.

Com a implementação desta técnica, conseguimos ganhos de desempenho significativos.

3.5 Otimizações das Transformações Estáticas

Durante o desenvolvimento do projeto, houve a necessidade de distinguir as transformações estáticas de transformações relativas a animações, em especifico para os casos das rotações e translações, onde esta última utiliza as curvas de Catmull-Rom para cálculo da posição seguinte ao longo de um dado tempo.

Para simplificar este processo, durante o processo de parsing, e com recurso à biblioteca glm, fomos multiplicando numa matriz local ao grupo, a matriz relativa à transformação que está a ser processada, garantindo que o resultado final, para um determinado group, é uma matriz com as transformações estáticas do xml aplicadas na ordem correta. Assim, no processo de desenho do group em questão, basta multiplicar a matriz do OpenGL pela matriz 'static_transformations', armazenada em cada group. Esta matriz tem como valor inicial a matriz identidade, para que caso não hajam transformações estáticas associadas a um dado group, esta multiplicação não anule as transformações já aplicadas pelo group-pai.

3.6 Curvas de Catmull-Rom

As curvas de **Catmull-Rom** foram introduzidas para permitir animações de translação suaves e contínuas entre um conjunto de pontos de controlo. Esta técnica é essencial para simular movimentos curvos, como o trajeto de planetas ou satélites no espaço tridimensional.

Uma curva de Catmull-Rom é uma curva paramétrica que passa diretamente pelos pontos de controlo fornecidos. A curva entre cada par de pontos é influenciada pelos dois pontos anteriores e seguintes, garantindo continuidade C^1 (derivada contínua), ideal para animações suaves.

A equação de uma curva de Catmull-Rom para um parâmetro $t \in [0, 1]$ e quatro pontos de controlo consecutivos P_0, P_1, P_2, P_3 é dada por:

$$P(t) = 0.5 \cdot \big(2P_1 + (-P_0 + P_2)t + (2P_0 - 5P_1 + 4P_2 - P_3)t^2 + (-P_0 + 3P_1 - 3P_2 + P_3)t^3\big) \tag{54}$$

No processo de parsing da cena, os pontos de controlo de uma curva são lidos do XML e armazenados para posterior interpolação. Durante a execução do programa, a posição ao longo da curva é calculada com base no tempo decorrido e no total de tempo especificado para a animação. Esse valor é então usado como o parâmetro t da equação, determinando a nova posição do objeto no espaço.

Para calcular também a orientação do objeto ao longo da curva, foi implementada a extração do vetor tangente $\vec{T}$ à curva em cada ponto. Este vetor é calculado derivando a equação de Catmull-Rom relativamente a t. O vetor tangente permite construir uma base local para aplicar a transformação `glRotatef`, garantindo que o objeto se orienta corretamente ao longo do percurso. Este vetor, apenas é aplicado à matriz de translação caso o parametro 'align' do XML esteja marcado como 'true'. Caso contrário, apenas uma translação é executada para a nova posição cálculada no polinómio.

Esta funcionalidade proporciona animações mais realistas e visualmente agradáveis, sendo essencial para cenas em que o movimento não é linear, como simulações astronómicas ou percursos complexos em ambientes 3D.

3.7 Sistema de Iluminação

A iluminação do sistema é implementada utilizando a API OpenGL clássica (OpenGL Fixed Function Pipeline), através da definição dos materiais de cada modelo. Esta abordagem permite simular como a luz interage com os objetos, conferindo-lhes um aspeto mais realista.

3.7.1 Definição do Material

Cada modelo possui um material associado, cujas propriedades são carregadas a partir do ficheiro XML e armazenadas numa estrutura do tipo `Material`. As propriedades definidas incluem:

- **Ambiente (`ambient`)**: representa a cor do material sob luz ambiente.
- **Difuso (`diffuse`)**: representa a cor refletida diretamente da fonte de luz.
- **Especular (`specular`)**: representa os brilhos especulares (reflexos brilhantes).
- **Emissivo (`emission`)**: define a cor que o objeto emite por si mesmo.
- **Brilho (`shininess`)**: controla a intensidade e dispersão do reflexo especular.

No momento da renderização (`Model::draw`), estas propriedades são aplicadas através da função `glMaterialfv` e `glMaterialf`, como se pode ver na função `setupMaterial()`:

```
glMaterialfv(GL_FRONT, GL_AMBIENT, glm::value_ptr(material.ambient));
glMaterialfv(GL_FRONT, GL_DIFFUSE, glm::value_ptr(material.diffuse));
glMaterialfv(GL_FRONT, GL_SPECULAR, glm::value_ptr(material.specular));
glMaterialfv(GL_FRONT, GL_EMISSION, glm::value_ptr(material.emission));
glMaterialf(GL_FRONT, GL_SHININESS, material.shininess);
```

Estas chamadas informam ao OpenGL como o material do objeto reage à luz, permitindo efeitos de iluminação como reflexos e sombreamento.

3.7.2 Configuração e Visualização das Luzes

A função `setupLights()` é responsável por configurar as fontes de luz da cena. Nesta função são definidos os parâmetros de cada luz, como posição, direção (no caso de luzes direcionais ou spotlights), cor ambiente, difusa e especular, bem como propriedades específicas como o ângulo de cutoff e a atenuação da luz.

Cada luz é associada a um identificador do OpenGL (`GL_LIGHT0`, `GL_LIGHT1`, etc.), que é ativado com `glEnable()`.

Além da configuração, o sistema inclui a função `drawLights()`, que é chamada durante a renderização para desenhar representações visuais (por exemplo, esferas ou pequenos cubos) nas posições das luzes, ajudando na compreensão espacial da cena. Esta função não afeta a iluminação, apenas serve como apoio visual para o programador ou utilizador.

Em conjunto, estas funções permitem uma gestão clara e modular das fontes de luz da cena 3D.

3.7.3 Normais e Iluminação

Para que a iluminação funcione corretamente, cada vértice do modelo deve ter um vetor normal associado. Estes vetores são carregados a partir do ficheiro do modelo (com a estrutura `x y z nx ny nz tx ty`) e armazenados num Vertex Buffer Object (VBO) específico para normais.

Durante a renderização, as normais são ativadas com:

```
glEnableClientState(GL_NORMAL_ARRAY);
glNormalPointer(GL_FLOAT, 0, 0);
```

Este passo é essencial, pois a OpenGL utiliza as normais para calcular como a luz incide sobre cada superfície.

3.7.4 Visualização das Normais

Para efeitos de debug ou demonstração, o sistema permite visualizar as normais de cada vértice. Quando a opção `viewNormals` está ativa, a função `drawNormals()` é chamada, desenhando linhas vermelhas a partir de cada vértice na direção da sua normal, com um comprimento reduzido.

3.8 Texturas

No desenvolvimento do projeto, uma das partes mais importantes foi a aplicação de texturas realistas aos modelos, como os planetas e suas luas. As texturas são essenciais para dar uma sensação de realismo visual à cena 3D, representando com precisão a aparência de cada modelo.

Para carregar e aplicar essas texturas, foi criada uma função chamada `loadTextures()`. Esta função é responsável por carregar todas as texturas necessárias para os modelos. Ela chama uma função auxiliar chamada `loadTexture(const char* filename)`, que é responsável pelo carregamento individual de cada imagem de textura a partir de um ficheiro de imagem.

Cada vez que a função `loadTexture` é chamada, ela usa a biblioteca devIL para carregar o conteúdo do arquivo de imagem em uma textura OpenGL. A biblioteca oferece suporte a formatos de imagem populares e cuida da conversão dessas imagens para um formato utilizável pelo OpenGL. A textura carregada é então associada a um identificador único (ID) que será utilizado em momentos posteriores do código para aplicar a textura ao objeto 3D correspondente.

A função `loadTextures()` carrega uma série de texturas para os diferentes modelos. Cada uma dessas texturas é armazenada em variáveis distintas para que possam ser facilmente acessadas durante a renderização da cena.

A função `loadTexture`, por sua vez, também configura as propriedades da textura, como o filtro de interpolação (linear) e o uso de mipmaps, que são versões reduzidas da textura original utilizadas para melhorar o desempenho e a qualidade visual ao renderizar objetos distantes. Além disso, a função garante que a origem da textura esteja invertida verticalmente, o que é necessário devido às diferenças entre as coordenadas de textura e a forma como as imagens são carregadas no OpenGL.

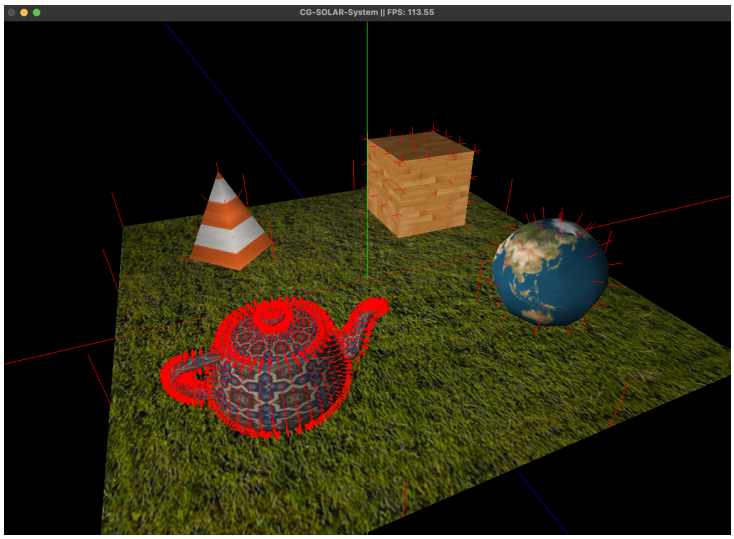


Figura 11: Cena de testes com iluminação, textura e visualização de normais ativada

4 Demo Final

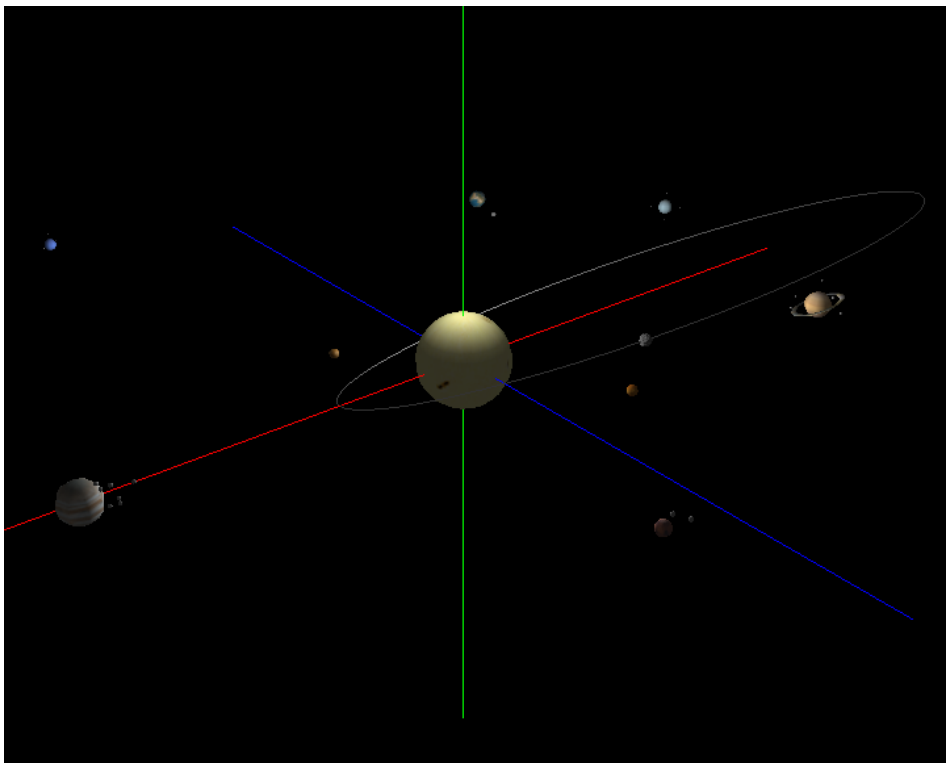


Figura 12: Demo final

5 Análise Crítica ao Projeto

A análise crítica de um projeto ou de uma implementação envolve uma reflexão detalhada sobre os pontos fortes e fracos do trabalho realizado, destacando aspectos técnicos, metodológicos e práticos. No caso deste projeto, a análise crítica abrange várias áreas, incluindo a concepção do sistema, a implementação das funcionalidades, e a interação com o ambiente de desenvolvimento e os utilizadores.

5.1 Objetivos do Projeto

Em primeiro lugar, é importante rever os objetivos definidos para o projeto. A missão inicial era **desenvolver um motor gráfico 3D, com recurso ao OpenGL para a criação de cenas 3D**, e a solução proposta teve como foco exatamente esse objetivo. A análise crítica começa por avaliar até que ponto os objetivos iniciais foram atingidos e se houve necessidade de ajustes ao longo do desenvolvimento. O projeto visava não só desenhar objetos 3D básicos, como também permitir interações dinâmicas com a cena, incluindo transformações, navegação e organização de múltiplos elementos gráficos.

5.2 Metodologia e Ferramentas Utilizadas

O uso de ferramentas e a metodologia aplicada desempenham um papel crucial no desenvolvimento do projeto. A escolha de C++ juntamente com a biblioteca OpenGL, e a utilização do CMake como sistema de construção, foi uma decisão baseada no desempenho, flexibilidade e compatibilidade com bibliotecas gráficas de baixo nível. O projeto seguiu uma abordagem incremental, com desenvolvimento modular das funcionalidades, permitindo testar e integrar cada componente de forma faseada. Esta escolha revelou-se eficaz, possibilitando uma boa gestão de complexidade.

5.3 Desafios Encontrados

Durante o desenvolvimento, diversos desafios técnicos e operacionais surgiram. Um dos maiores desafios foi a gestão de múltiplos triângulos e transformações aplicadas a cada um, o que exigiu a criação de uma arquitetura orientada a objetos mais sólida. Foi necessário desenhar uma estrutura que encapsulasse as transformações e o desenho dos triângulos, mantendo a flexibilidade para expansões que mais tarde foram concretizadas, como animações, iluminação e texturas. Esta abordagem obrigou à reformulação de parte do código e à reorganização do pipeline de desenho.

5.4 Aspetos Positivos

Entre os pontos fortes do projeto, destaca-se a organização modular do código, a facilidade de extensão da cena gráfica e a correta separação entre lógica

de transformação e renderização. Estes elementos foram atingidos através da criação de classes específicas para entidades gráficas e da utilização eficiente do OpenGL para desenhar em tempo real. Além disso, a integração com CMake facilitou a compilação cruzada e a portabilidade do projeto entre diferentes plataformas.

De se destacar ainda as otimizações no que toca ao desempenho (com o recurso a VBOs) e à visualização da cena criada, com a possibilidade de análise da cena através de uma camara orbital e opções de visualização pertinentes para uma análise mais detalhada da cena.

5.5 Aspetos Negativos

Por outro lado, existem áreas que poderiam ser melhoradas. Um ponto negativo é a ausência de um sistema de câmara mais completo, com movimentação livre ou controlo de perspetiva, o que limita a navegação na cena. Para além disso, a falta de uma interface gráfica para controlo de parâmetros em tempo real (como cores, posições ou visibilidade de objetos) reduz a interatividade e a capacidade de demonstração do motor.

No que toca ao desempenho, faltou ainda completar a implementaçao de VBOs com a funcionalidade de indices, o que permitia o ganho marginal de frames por segundo na cena e uma maior economia de memória utilizada.

5.6 Soluções Propostas

Em relação aos aspetos negativos, algumas soluções podem ser propostas para melhorias futuras. Uma possível solução para a limitação da navegação na cena seria implementar um sistema de câmara com controlo por teclado e rato, utilizando matrizes de visualização e projeção mais avançadas. Estas adições tornariam o motor mais versátil e amigável para o utilizador.

5.7 Conclusão da Análise Crítica

A análise crítica deve, finalmente, concluir sobre o sucesso do projeto, considerando tanto as suas forças como as suas fraquezas. Em última instância, o projeto cumpriu os seus objetivos iniciais e grande parte dos extras propostos, demonstrando uma aplicação prática e funcional de um motor gráfico 3D com recurso ao OpenGL. Embora haja aspetos a melhorar, especialmente no que diz respeito à interação com o utilizador e à navegabilidade da cena, as bases estruturais estão bem definidas e prontas para expansão. As lições aprendidas ao longo do desenvolvimento, tanto a nível técnico como organizacional, serão certamente valiosas para futuros projetos nesta área.

6 Referências

- https://pages.mtu.edu/ shene/COURSES/cs3621/NOTES/surface/bezier-construct.html
- https://www.songho.ca/opengl/gl$_t$*orus.html*
- *https* : *//www.songho.ca/opengl/gl$_c$ylinder.html*
- *https* : *//www.songho.ca/opengl/gl$_s$phere.html*
- *https* : *//www.songho.ca/opengl/gl$_c$one.html*
- Slides das aulas TPs

7 Anexos

Nesta secção, deixamos em anexo imagens relativas à evolução da demo do sistema solar, desde a primeira fase até ao final, como forma de demonstrar a evolução do projeto.

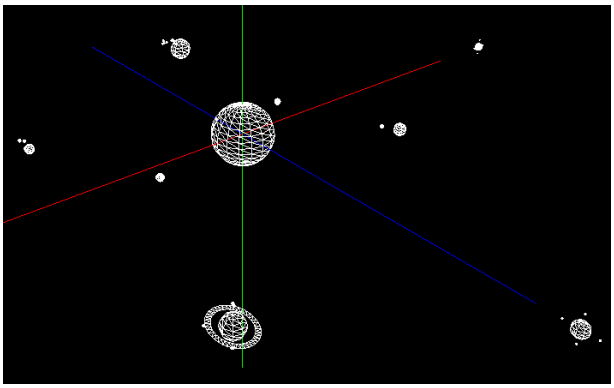


Figura 13: Demo fase 2

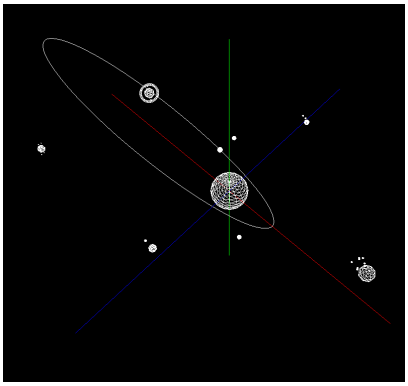


Figura 14: Demo fase 3

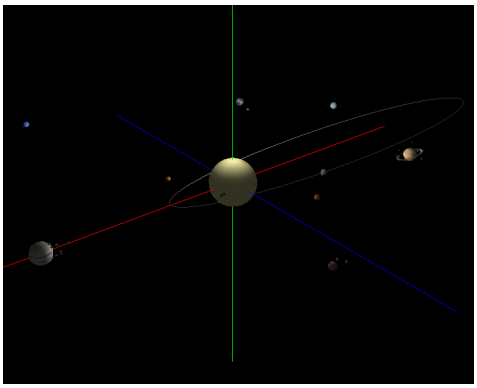


Figura 15: Demo final