

# Relatório Técnico

## Practical Assignment

### Trabalho Prático

Computação Gráfica



## Universidade do Minho

### Universidade do Minho

Escola de Engenharia, Braga, Portugal  
Licenciatura em Engenharia Informática

Ano Letivo 2025/2026

### Grupo 25

Carlos Silva; Francisco Veloso; Nuno Soares; Duarte Faria

a106792@uminho.pt

a106882@uminho.pt

a107366@uminho.pt

a95609@uminho.pt

# Conteúdo

1	Introdução .....	3
2	Fase 1: Primitivas e Motor de Visualização .....	4
2.1	Fluxo de Dados .....	4
2.2	Generator — Geração de Primitivas .....	4
2.2.1	Plano (Plane) .....	4
2.2.2	Algoritmo .....	4
2.2.3	Caixa (Box) .....	5
2.2.4	Algoritmo .....	5
2.2.5	Esfera (Sphere) .....	5
2.2.6	Algoritmo .....	6
2.2.7	Cone .....	6
2.2.8	Geometria do Cone .....	7
2.2.9	Algoritmo .....	7
2.3	Engine — Motor Gráfico .....	7
2.3.1	Formato do Ficheiro .3d .....	7
2.3.2	Leitura da Configuração XML .....	8
2.3.3	Pipeline de Renderização .....	8
2.3.4	Callback de Reshape .....	8
2.3.5	Callback de Display .....	8
2.3.6	Sistema de Câmera .....	9
2.3.7	Câmera Fixa (XML) .....	9
2.3.8	Câmera Orbital (Explorer) .....	9
2.3.9	Modos de Renderização .....	9
2.3.10	Controlos Adicionais .....	9
2.4	Resultados obtidos .....	10
2.5	Conclusão .....	12

## 1 Introdução

O presente relatório descreve o desenvolvimento da primeira fase do Trabalho Prático da unidade curricular de Computação Gráfica. O objetivo principal desta fase consistiu na construção da base de um motor gráfico 3D, recorrendo à biblioteca OpenGL/GLUT, e na implementação de um sistema modular composto por duas aplicações distintas: um *generator*, responsável pela criação das primitivas geométricas, e um *engine*, encarregado da leitura de ficheiros de configuração e da renderização da cena.

Ao longo do trabalho, implementámos as várias primitivas tridimensionais, diferentes modos de visualização e um sistema de câmara configurável, cumprindo assim a estrutura fundamental que servirá de base para as fases seguintes do projeto que o nosso grupo implementará.

## 2 Fase 1: Primitivas e Motor de Visualização

O objetivo da Fase 1 é construir a base do motor 3D que será desenvolvido ao longo das quatro fases do trabalho. Esta fase foca-se em dois componentes fundamentais:

1. **Generator** — aplicação independente que gera ficheiros `.3d` contendo os vértices das primitivas geométricas.
2. **Engine** — aplicação que lê um ficheiro XML de configuração (câmara, janela, modelos) e renderiza a cena com OpenGL/GLUT.

A separação em dois programas distintos permite que a computação pesada da geometria seja feita apenas uma vez.

### 2.1 Fluxo de Dados

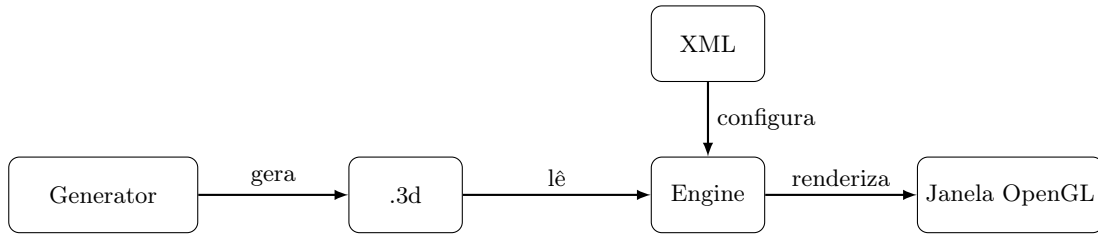


Figura 1: Fluxo de dados entre os componentes.

### 2.2 Generator — Geração de Primitivas

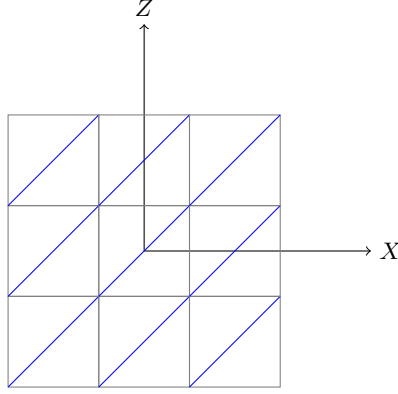
A abordagem adotada no *generator* baseia-se em superfícies paramétricas. Uma decisão fundamental foi o uso do *Counter-Clockwise Winding Order* para todos os triângulos, garantindo que as faces externas sejam corretamente renderizadas pelo OpenGL com *Back-face Culling* ativo. Ao definir os vértices nesta ordem, a face frontal de cada polígono é determinada pela regra da mão direita, permitindo que o *engine* descarte a renderização de faces internas ou não visíveis.

#### 2.2.1 Plano (Plane)

O plano é gerado no plano  $XZ$  ( $y = 0$ ), centrado na origem, com um dado comprimento  $L$  e número de subdivisões  $D$ .

#### 2.2.2 Algoritmo

1. Calcular o tamanho de cada subdivisão:  $s = L/D$
2. Calcular o offset de centralização:  $h = L/2$
3. Para cada célula  $(i, j)$  da grelha  $D \times D$ :
  - Calcular os 4 cantos:  $(x_1, z_1) = (-h + i \cdot s, -h + j \cdot s)$  e  $(x_2, z_2) = (-h + (i+1) \cdot s, -h + (j+1) \cdot s)$
  - Emitir triângulo 1:  $(x_1, 0, z_1), (x_1, 0, z_2), (x_2, 0, z_2)$
  - Emitir triângulo 2:  $(x_1, 0, z_1), (x_2, 0, z_2), (x_2, 0, z_1)$



Plano com 3 divisões (vista de cima)

Figura 2: Subdivisão do plano. A diagonal azul indica a divisão de cada quad em dois triângulos.

### 2.2.3 Caixa (Box)

A caixa é centrada na origem com um dado tamanho  $S$  e número de divisões  $D$  por aresta. Cada uma das 6 faces é uma grelha de  $D \times D$  quads, cada quad dividido em 2 triângulos.

### 2.2.4 Algoritmo

1. Calcular:  $h = S/2$  (metade do tamanho),  $s = S/D$  (passo da subdivisão)
2. Para cada uma das 6 faces (frente, trás, direita, esquerda, cima, baixo):
  - Para cada célula  $(i, j)$  da grelha  $D \times D$ :
    - Calcular os 4 cantos do quad na face
    - Emitir 2 triângulos com *winding* CCW (visto de fora)

**Decisão de projeto:** A normal de cada face aponta para fora da caixa. Os triângulos são emitidos com ordem CCW quando vistos do exterior, garantindo compatibilidade com *back-face culling*.

O número total de vértices para a caixa é:

$$V_{\text{box}} = 6 \times D^2 \times 2 \times 3 = 36 \cdot D^2$$

### 2.2.5 Esfera (Sphere)

A esfera é gerada a partir da conversão de coordenadas esféricas para coordenadas cartesianas. No código, adotamos a convenção em que o ângulo  $\theta$  representa a componente polar e o ângulo  $\phi$  a componente azimutal.

**Equações Paramétricas:**

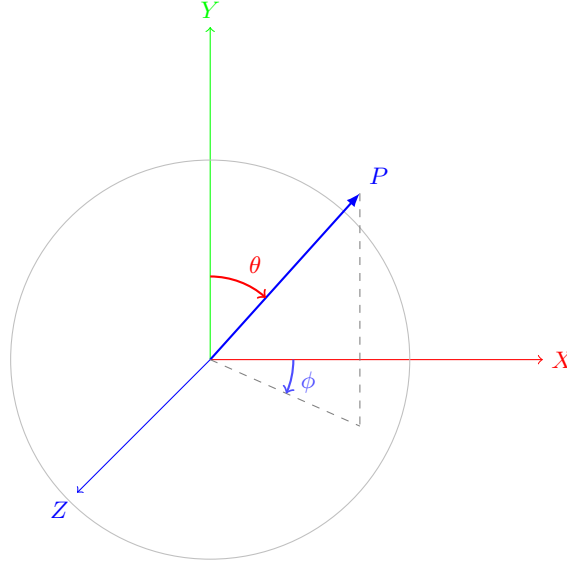
$$\begin{cases} x = r \cdot \sin(\theta) \cdot \sin(\phi) \\ y = r \cdot \cos(\theta) \\ z = r \cdot \sin(\theta) \cdot \cos(\phi) \end{cases} \quad (1)$$

A utilização direta de  $\cos(\theta)$  no eixo  $Y$  mostra que este é o eixo vertical da esfera, ficando assim alinhado com o sistema de coordenadas padrão do OpenGL.

Durante a implementação, tivemos o cuidado de tratar corretamente os polos da esfera. No topo ( $\theta = 0$ ) e na base ( $\theta = \pi$ ), todos os pontos de cada *slice* coincidem no mesmo local. Se gerássemos quadriláteros nessas zonas (ou seja, dois triângulos por divisão), iríamos obter triângulos degenerados, com área nula.

Para evitar esse problema, o algoritmo identifica as camadas correspondentes aos polos e, nessas situações, gera apenas um triângulo por *slice*, formando um *triangle fan* que converge no polo. Desta forma, reduzimos o número de vértices desnecessários, tornando o processo ligeiramente mais eficiente.

A superfície da esfera é dividida em *slices*  $S_l$  (divisões ao longo do ângulo azimutal) e *stacks*  $S_t$  (divisões ao longo do ângulo polar). Cada quadrilátero formado entre duas *stacks* e duas *slices* é dividido em dois triângulos através da diagonal que liga o vértice  $v_1$  ao vértice  $v_3$ .



$\theta$ : ângulo polar,  $\phi$ : ângulo azimutal

Figura 3: Sistema de coordenadas esféricas utilizado na geração da Esfera.

### 2.2.6 Algoritmo

1. Para cada stack  $i$  ( $0 \leq i < S_t$ ):
  - Calcular  $\theta_1 = \pi \cdot i / S_t$  e  $\theta_2 = \pi \cdot (i+1) / S_t$
2. Para cada slice  $j$  ( $0 \leq j < S_l$ ):
  - Calcular  $\phi_1 = 2\pi \cdot j / S_l$  e  $\phi_2 = 2\pi \cdot (j+1) / S_l$
  - Calcular os 4 vértices:  $v_1(\theta_1, \phi_1)$ ,  $v_2(\theta_2, \phi_1)$ ,  $v_3(\theta_2, \phi_2)$ ,  $v_4(\theta_1, \phi_2)$
  - Se  $i \neq S_t - 1$ : emitir triângulo  $v_1, v_2, v_3$  (não degenera no polo sul)
  - Se  $i \neq 0$ : emitir triângulo  $v_1, v_3, v_4$  (não degenera no polo norte)

**Tratamento dos polos:** No polo norte ( $\theta = 0$ ), os vértices  $v_1$  e  $v_4$  coincidem no ponto  $(0, r, 0)$ , pelo que o triângulo  $v_1 v_3 v_4$  degenera e é omitido. No polo sul ( $\theta = \pi$ ),  $v_2$  e  $v_3$  coincidem em  $(0, -r, 0)$ , pelo que o triângulo  $v_1 v_2 v_3$  degenera e é omitido.

O número total de triângulos é:

$$T_{\text{sphere}} = 2 \cdot S_l \cdot (S_t - 2) + 2 \cdot S_l = 2 \cdot S_l \cdot (S_t - 1)$$

### 2.2.7 Cone

O cone é gerado com a base no plano  $XZ$  ( $y = 0$ ), centrado na origem. O raio diminui linearmente com a altura, de  $R$  (na base) a 0 (no vértice).

### 2.2.8 Geometria do Cone

Para uma stack  $i$  ( $0 \leq i < S_t$ ), o raio e a altura são:

$$r_i = R \left( 1 - \frac{i}{S_t} \right) \quad (2)$$

$$y_i = h \cdot \frac{i}{S_t} \quad (3)$$

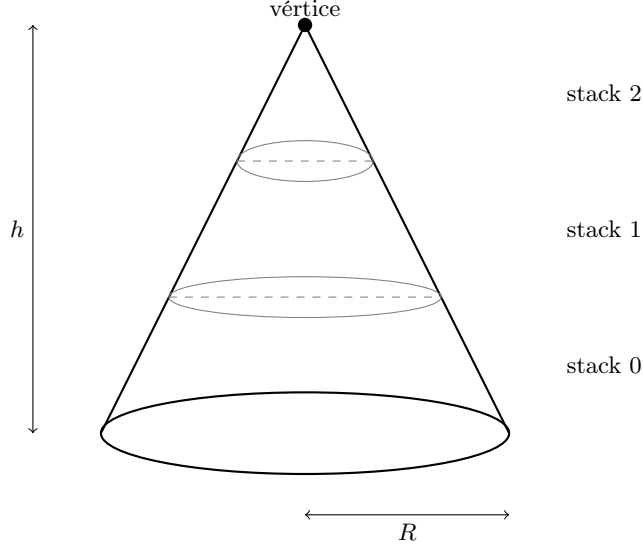


Figura 4: Estrutura do cone com 3 stacks.

### 2.2.9 Algoritmo

1. **Base** — leque (fan) de triângulos no plano  $XZ$ :
  - Para cada *slice*  $j$ , é emitido um triângulo com centro em  $(0,0,0)$  e dois pontos consecutivos da circunferência da base (ordem CCW quando observado de baixo, com a normal orientada segundo  $-Y$ ).
2. **Corpo** — dividido em *stacks*:
  - Para cada *stack*  $i$  e *slice*  $j$ :
    - Calculam-se os quatro vértices  $v_1, v_2, v_3, v_4$  que definem o *quad*;
    - Emite-se o triângulo  $v_1, v_3, v_2$  (ordem CCW quando observado do exterior);
    - Se  $i \neq S_t - 1$  (ou seja, se não for a última *stack*), emite-se também o triângulo  $v_1, v_4, v_3$ .

Na última stack, os vértices superiores convergem no vértice do cone, pelo que apenas é emitido um triângulo por slice.

## 2.3 Engine — Motor Gráfico

### 2.3.1 Formato do Ficheiro .3d

O formato de ficheiro adotado para armazenar os modelos é simples e textual:

- **Linha 1:** número total de vértices  $N$

- **Linhas 2 a  $N + 1$ :** coordenadas  $x y z$  de cada vértice (um por linha)

Os vértices são armazenados em grupos de 3 (formando triângulos), na ordem em que devem ser desenhados com `GL_TRIANGLES`. O *winding order* é **counter-clockwise** (CCW), que é o default do OpenGL para *front faces*.

### 2.3.2 Leitura da Configuração XML

Usámos a biblioteca *TinyXML-2* para interpretar o ficheiro de cena já que foi a que nos foi recomendada. A decisão de usar XML permite configurar a resolução da janela e os parâmetros da câmara sem necessidade de recompilar o código, garantindo flexibilidade na montagem de diferentes cenários.

O engine recebe como argumento um ficheiro XML que define:

- **Janela:** dimensões (largura, altura)
- **Câmara:** posição, ponto de foco (*lookAt*), vetor *up*, e parâmetros de projeção (FOV, *near*, *far*)
- **Modelos:** lista de ficheiros *.3d* a carregar

Alguns Valores por omissão são definidos para os campos opcionais (**up** = (0,1,0), **fov** = 60, **near** = 1, **far** = 1000), conforme especificado na sintaxe do file XML de referência colocado na blackboard.

### 2.3.3 Pipeline de Renderização

A renderização segue o pipeline clássico do OpenGL com GLUT:

1. **Inicialização:** `glutInit`, configuração do modo de display (RGBA, double buffer, depth buffer), criação da janela.
2. **Configuração OpenGL:** ativação do *depth test* e *back-face culling*.
3. **Registo de callbacks:** display, reshape, keyboard.
4. **Ciclo principal:** `glutMainLoop`.

### 2.3.4 Callback de Reshape

No redimensionamento da janela:

1. Ativar a matriz de projeção
2. Resetar a matriz
3. Configurar o *viewport* e aplicar a projeção perspetiva com os parâmetros do XML
4. Voltar à matriz de *modelview*

### 2.3.5 Callback de Display

A cada frame:

1. Limpar buffers de cor e profundidade
2. Posicionar a câmara com `gluLookAt`
3. Definir o modo de polígono (*solid/wireframe/points*)
4. Desenhando eixos de referência (com *depth test* ativo)
5. Desenhando os modelos com triângulos (modo imediato)
6. Trocar buffers (*double buffering*)

**Nota:** Desenhámos os eixos **antes** dos modelos, mantendo o *depth test* ativo em ambos os casos. Isto garante que a geometria dos modelos que está mais próxima da câmara se sobrepõe naturalmente aos eixos, o que respeita corretamente a profundidade.

Inicialmente, não tínhamos implementado desta forma e encontrámos alguns problemas na renderização das figuras. Embora a figura fosse estruturalmente igual à dos testes, algumas linhas que evidenciavam a profundidade não eram renderizadas corretamente.



### 2.3.6 Sistema de Câmera

Foram implementados dois modos de câmera:

### 2.3.7 Câmera Fixa (XML)

Usa diretamente os valores de posição, *lookAt* e *up* definidos no ficheiro XML.

### 2.3.8 Câmera Orbital (Explorer)

Implementou-se uma câmara orbital baseada em coordenadas esféricas para facilitar a inspeção dos modelos. O utilizador controla os ângulos  $\alpha$  (rotação horizontal) e  $\beta$  (inclinação vertical).

---

**Algorithm 1** Conversão para Câmera Orbital

---

```
1:  $pos.x \leftarrow raio \cdot \cos(\beta) \cdot \sin(\alpha)$   
2:  $pos.y \leftarrow raio \cdot \sin(\beta)$   
3:  $pos.z \leftarrow raio \cdot \cos(\beta) \cdot \cos(\alpha)$ 
```

---

onde  $\alpha$  é o ângulo azimutal,  $\beta$  é a elevação (limitada a  $\pm 1.5$  rad  $\approx \pm 86$  para evitar *gimbal lock*), e  $r$  é a distância ao foco (Q/E).

A câmera orbital é inicializada a partir da posição XML, convertendo coordenadas cartesianas para esféricas:

$$r = \sqrt{p_x^2 + p_y^2 + p_z^2} \quad (4)$$

$$\alpha = \text{atan2}(p_x, p_z) \quad (5)$$

$$\beta = \arcsin\left(\frac{p_y}{r}\right) \quad (6)$$

### 2.3.9 Modos de Renderização

Três modos são suportados:

Tecla	Modo	Descrição
1	Sólido	Faces preenchidas
2	Wireframe	Apenas arestas
3	Pontos	Apenas vértices

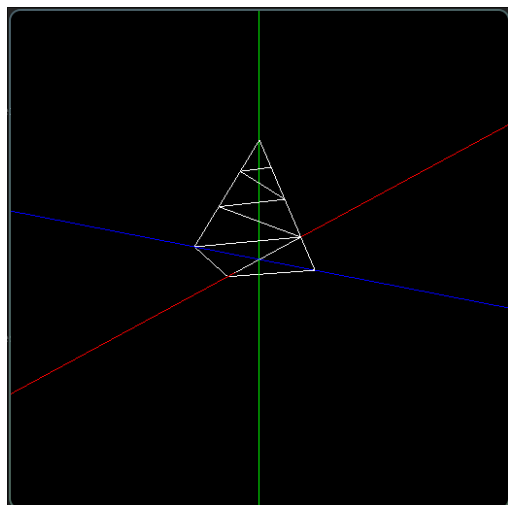
Tabela 1: Modos de renderização disponíveis.

### 2.3.10 Controlos Adicionais

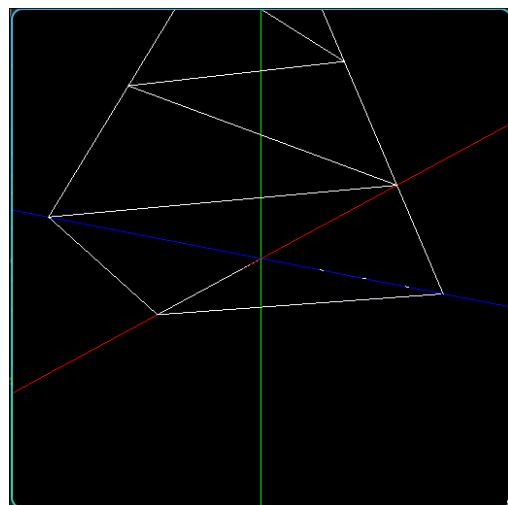
Tecla Ação	
W/S	Rodar câmera verticalmente (elevação)
A/D	Rodar câmera horizontalmente (azimute)
Q/E	Zoom in/out (alterar raio)
C	Alternar câmera fixa/orbital
ESC	Sair da aplicação

Tabela 2: Controlos do engine.

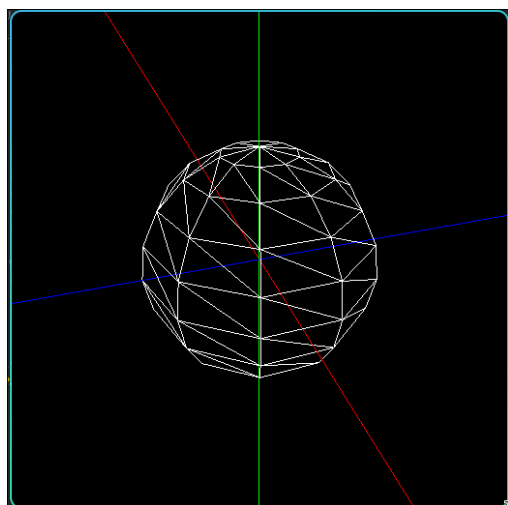
## 2.4 Resultados obtidos



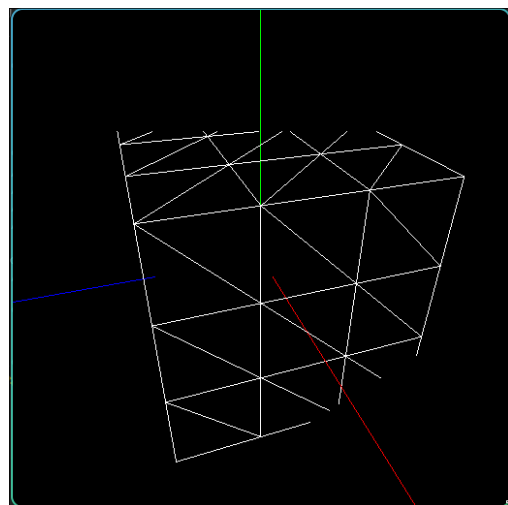
(a) Teste 1



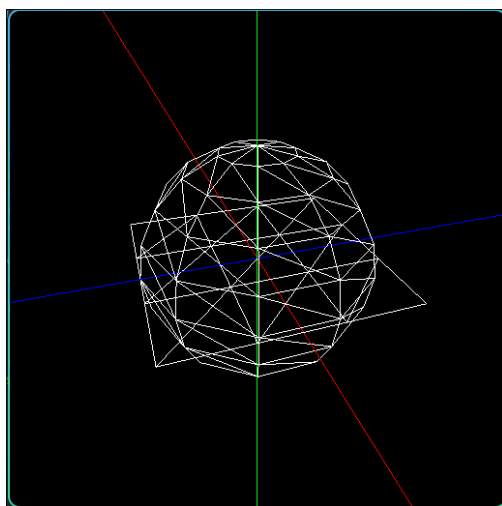
(b) Teste 2



(c) Teste 3



(d) Teste 4



(e) Teste 5

Figura 5: Resultados dos testes

## 2.5 Conclusão

Nesta primeira fase, implementámos corretamente as quatro primitivas que nos foram solicitadas no generator, bem como o engine capaz de ler as configurações em XML e renderizar os modelos com OpenGL/GLUT, tal como praticámos nas aulas. O nosso projeto inclui algumas configurações de câmaras, como a fixa e a orbital, três modos de renderização e os respetivos eixos de referência.

De uma forma geral, o nosso grupo sente que conseguiu realizar, sem grandes dificuldades, o que nos foi proposto para esta primeira fase de entrega. O único detalhe que demorou um pouco mais a ser concretizado foi a confirmação e correção da imagem produzida pelo nosso engine, de modo a que fosse exatamente igual à fornecida na diretoria com as imagens dos resultados esperados. A parte de gerar as figuras decorreu sem problemas. No entanto, garantir que ficassem exatamente iguais às imagens de teste exigiu mais algum tempo.

## Referências

1. Não foram utilizadas referências bibliográficas externas. O trabalho foi desenvolvido com base nos slides teóricos, práticos e os guíões lecionados na unidade curricular.