Universidade do Minho

Licenciatura em Engenharia Informática

Computação Gráfica

Trabalho Prático- Fase 1

TP16

Trabalho realizado por:
Eduarda Mafalda Martins Vieira a104098

Maria Leonor Carvalho da Cunha a103997

Pedro Manuel Macedo Rebelo a104091

fevereiro de 2026

Índice

1 Introdução

No âmbito da unidade curricular de Computação Gráfica foi-nos pedido o desenvolvimento de um projeto de representação gráfica 3D baseado numa arquitetura de motor de cenas (scene graph engine).

Este projeto foi estruturado em quatro fases de entrega, sendo que a primeira (atual) consiste em desenvolver as bases arquitectónicas do sistema. Nesta fase foram implementadas duas aplicações complementares:

- **Generator**: Aplicação responsável por gerar ficheiros contendo as definições das primitivas geométricas através de conjuntos de vértices.
- **Engine**: Aplicação responsável por ler ficheiros de configuração XML e renderizar as cenas utilizando as primitivas previamente geradas.

Nesta primeira fase implementámos as seguintes primitivas geométricas: plano, caixa (box), esfera e cone. Todas as primitivas foram desenvolvidas com suporte a subdivisões, permitindo controlar o nível de detalhe de cada modelo.

O projeto foi desenvolvido em linguagem **C++** recorrendo à biblioteca **OpenGL** para a renderização gráfica e à biblioteca **tinyxml2** para o processamento de ficheiros de configuração em formato XML.

1.1 Requisitos da Fase 1

Com base no enunciado, verificamos os seguintes requisitos:

- **Duas aplicações** (Generator + Engine): implementado.
- **Primitivas exigidas** (plano, box, esfera, cone): implementadas.
- **Centro/orientação**: plano e box centrados na origem, esfera centrada na origem, cone com base no plano XZ (y=0): implementado.
- **Gerador por linha de comando** (tipo + parâmetros + ficheiro .3d): implementado.
- **Leitura do XML apenas no arranque** (câmara + modelos): implementado.
- **Estruturas em memória** para modelos: implementado.
- **Parser XML** com tinyxml2: implementado.

2 Arquitetura do Projeto

2.1 Aplicações

O projeto está organizado em duas aplicações independentes que se complementam:

2.1.1 Generator

O módulo generator.cpp é responsável por converter primitivas geométricas em conjuntos de vértices armazenados em ficheiros com exten- são .3d. Esta aplicação funciona de forma autónoma, recebendo como parâmetros:

- O tipo de primitiva a gerar
- Os parâmetros específicos de cada primitiva
- O nome do ficheiro de destino

Os ficheiros gerados são armazenados na pasta files3d/ e contêm a definição das primitivas em formato XML estruturado, onde cada triângulo é representado pelos seus três vértices com coordenadas (x, y, z).

2.1.2 Engine

O módulo engine.cpp é responsável por:

1. Ler ficheiros de configuração XML que especificam a câmara, a janela, e os modelos a renderizar
2. Carregar os ficheiros .3d previamente gerados pelo Generator
3. Renderizar a cena utilizando OpenGL

O engine oferece uma interface interativa permitindo ao utilizador: - Visualizar os modelos 3D - Ativar/desativar modo wireframe - Mostrar/esconder eixos de coordenadas - Interagir através de teclado com a cena

2.2 Classes

De modo a facilitar a implementação das funcionalidades foram criadas as seguintes classes de suporte:

2.2.1 Point

A classe Point representa um ponto 3D no espaço cartesiano. Embora não seja explicitamente declarada como classe, é utilizada em estruturas como Vertex que armazenam três coordenadas float (X, Y, Z) representando um ponto no espaço.

2.2.2 Shape

A classe Shape (embora não seja explicitamente uma classe base) é o conceito utilizado para agrupar as funcionalidades de geração de primitivas. Cada primitiva (plano, box, esfera e cone) é gerada através de funções dedicadas que implementam a lógica matemática específica.

2.2.3 Camera

A classe Camera é responsável por todas as operações relacionadas com a câmara virtual. Armazena:

- **Posição**: Coordenadas (posX, posY, posZ) da câmara
- **LookAt**: Ponto para o qual a câmara está orientada (lookAtX, lookAtY, lookAtZ)
- **Up Vector**: Vetor que define a orientação vertical (upX, upY, upZ)
- **Projeção**: Parâmetros de projeção perspectiva (fov, nearPlane, farPlane)
- **Coordenadas Esféricas**: Suporte para manipulação da câmara (alpha, beta, radius)

A câmara fornece métodos getter e setter para manipular estes parâmetros, permitindo a implementação de transformações de visualização no pipeline gráfico do OpenGL.

2.2.4 Parser

A classe SimpleParser utiliza a biblioteca tinyxml2 para processar ficheiros XML de configuração. É responsável por:

1. Ler e interpretar ficheiros XML
2. Extrair informações sobre a janela (width, height)
3. Processar configurações da câmara
4. Parsing da lista de modelos a carregar

A classe armazena: - Window: std::pair<int, int> com as dimensões - Camera: Ponteiro para a câmara configurada - Group: Vetor de modelos a serem carregados

2.3 Ferramentas Utilizadas

- **OpenGL**: Framework principal para renderização gráfica 3D. Permite desenhar primitivas, aplicar transformações e gerir o pipeline gráfico.
- **GLUT**: Utilizado para criar janelas, gerir eventos de teclado/rato, e lidar com callbacks de renderização.
- **tinyxml2**: Biblioteca rápida e leve para parsing de ficheiros XML, utilizada para ler configurações.
- **C++ Standard Library**: Utilização de std::vector, std::string, std::fstream para estruturas de dados e I/O de ficheiros.
- **CMake**: Sistema de compilação para organizar o projeto (ficheiro CMake-Cache.txt presente).

3 Primitivas Geométricas

3.1 Plano

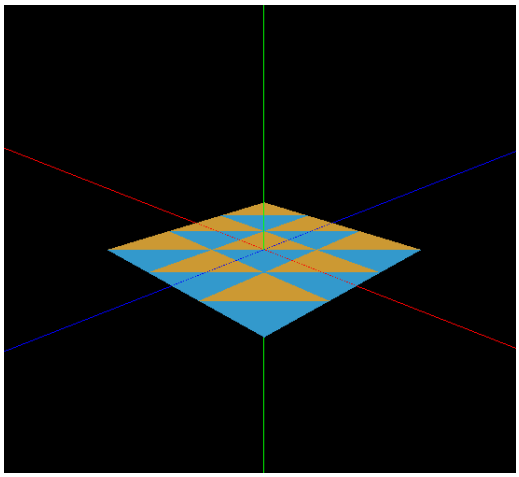

Figura 1 - plane

O plano representa uma superfície 2D rectangular no espaço XZ, centrada na origem, com altura Y = 0.

Parâmetros: - length: Comprimento do lado do plano - divisions: Número de subdivisões por eixo (cria uma malha divisions × divisions)
Algoritmo:

1. Calcula o tamanho de cada quadrado: step = length / divisions

2. Define o ponto de partida: start = -length / 2 (para centrar na origem)
3. Para cada quadrado da malha, gera dois triângulos: - Triângulo 1: vértices nos cantos inferior-esquerdo, superior-esquerdo, inferior-direito - Triângulo 2: vértices nos cantos inferior-direito, superior-esquerdo, superior-direito

Exemplo de vértices gerados (length=1, divisions=2):

(-0.5, 0, -0.5), (-0.5, 0, 0), (0, 0, -0.5)
(0, 0, -0.5), (-0.5, 0, 0), (0, 0, 0)

3.2 Box (Cubo)

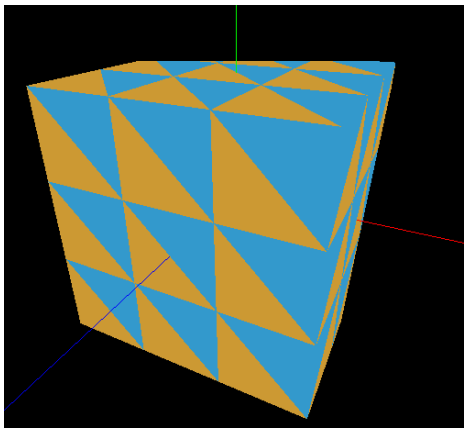

Figura 2 - box

A caixa ou cubo é uma extensão do plano, combinando 6 planos para formar um sólido 3D centrado na origem.

Parâmetros: - size: Dimensão da aresta do cubo - divisions: Número de subdivisões por face
Algoritmo:

1. Calcula halfSize = size / 2 e step = size / divisions
2. Gera 6 faces, cada uma sendo um plano subdivido: - Face frontal (Z = +halfSize) - Face traseira (Z = -halfSize) - Face direita (X = +halfSize) - Face esquerda (X = -halfSize) - Face superior (Y = +halfSize) - Face inferior (Y = -halfSize) 3. Para cada face, aplica as mesmas subdivisões do plano

Características: - Centrado na origem - Permite especificar o nível de detalhe independentemente para cada face - Facilita a aplicação de texturas diferentes por face (em fases posteriores)

3.3 Esfera

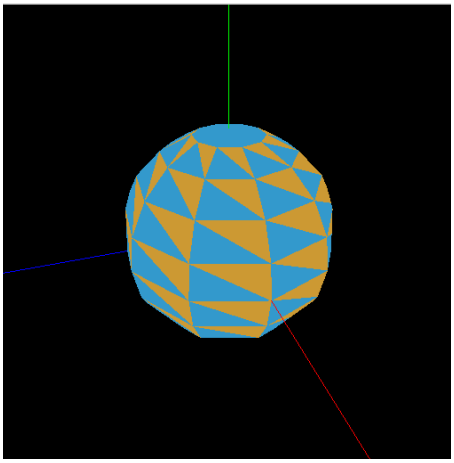

Figura 3 - sphere

A esfera é uma primitiva 3D que utiliza coordenadas esféricas para gerar uma superfície de revolução.

Parâmetros: - radius: Raio da esfera - slices: Número de fatias verticais (divisões azimutais) - stacks: Número de camadas horizontais (divisões polares)

Algoritmo com Coordenadas Esféricas:

1. Define incrementos angulares:
   - $\Delta\alpha = \frac{2\pi}{slices}$ (incremento azimutal)
   - $\Delta\beta = \frac{\pi}{stacks}$ (incremento polar)
2. Para cada camada *i* (de 0 a stacks-1) e cada fatia *j* (de 0 a slices-1):
   - $\beta_1 = i \cdot \Delta\beta$
   - $\beta_2 = (i+1) \cdot \Delta\beta$
   - $\alpha_1 = j \cdot \Delta\alpha$
   - $\alpha_2 = (j+1) \cdot \Delta\alpha$

Conversão de Coordenadas Esféricas para Cartesianas:

Para um ponto na esfera definido por (r , α, β):

$$P = \begin{pmatrix} r\sin(\beta)\cos(\alpha) \\ r\cos(\beta) \\ r\sin(\beta)\sin(\alpha) \end{pmatrix}$$

onde: - $\alpha \in [0, 2\pi]$ é o ângulo azimutal (rotação em torno do eixo Y) - β pertence $[0, \pi]$ é o ângulo polar (elevação a partir do polo norte)

Vértices dos Triângulos:

$$P_1 = r(\sin\beta_1 \cos\alpha_1, \cos\beta_1, \sin\beta_1 \sin\alpha_1)$$

$$P_2 = r(\sin\beta_1 \cos\alpha_2, \cos\beta_1, \sin\beta_1 \sin\alpha_2)$$

$$P_3 = r(\sin\beta_2 \cos\alpha_1, \cos\beta_2, \sin\beta_2 \sin\alpha_1)$$

$$P_4 = r(\sin\beta_2 \cos\alpha_2, \cos\beta_2, \sin\beta_2 \sin\alpha_2)$$

Vantagens: - Esfera suave com número de vértices controlável - Maior número de slices/stacks = aproximação melhor à esfera perfeita - Distribuição uniforme de vértices na superfície.

3.4 Cone

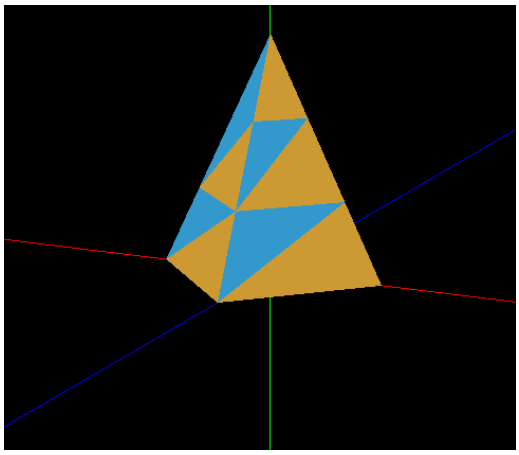

Figura 4 - cone

O cone é uma primitiva que combina uma base circular **fechada** com lados que convergem para um vértice no topo.

Parâmetros: - radius: Raio da base circular - height: Altura do cone - slices: Número de lados (divisões na base circular) - stacks: Número de camadas verticais nos lados

Algoritmo:

1. Cálculo de parâmetros:

- $\Delta\theta = \frac{2\pi}{slices}$ (ângulo entre fatias)
- $\Delta h = \frac{height}{stacks}$ (altura de cada camada)

2. Geração dos Lados (Surface):

Para cada camada i (de 0 a stacks-1) e cada fatia j (de 0 a slices-1):

- Raio na camada i: $r_i = radius \cdot \left(1 - \frac{i}{\text{stacks}}\right)$
- Raio na camada $i+1$: $r_{i+1} = radius \cdot \left(1 - \frac{i+1}{\text{stacks}}\right)$
- Altura na camada i: $y_i = i \cdot \Delta h$
- Altura na camada $i+1$: $y_{i+1} = (i+1) \cdot \Delta h$

Ângulos:

- $\theta_j = j \cdot \Delta\theta$
- $\theta_{j+1} = (j+1) \cdot \Delta\theta$

Vértices da camada lateral:

$$P_{i,j} = (r_i \cos\theta_j, y_i, r_i \sin\theta_j)$$

$$P_{i,j+1} = (r_i \cos\theta_{j+1}, y_i, r_i \sin\theta_{j+1})$$

$$P_{i+1,j} = (r_{i+1} \cos\theta_j, y_{i+1}, r_{i+1} \sin\theta_j)$$

$$P_{i+1,j+1} = (r_{i+1} \cos\theta_{j+1}, y_{i+1}, r_{i+1} \sin\theta_{j+1})$$

Caso especial (topo do cone, quando $i = \text{stacks} - 1$):

$$P_{\text{topo}} = (0, height, 0)$$

5. Geração da Base (Fechada):

A base é um disco circular no plano XZ ($y = 0$) dividido em triângulos radiantes.

Para cada fatia j (de 0 a slices-1):

- Centro: $C = (0, 0, 0)$
- Ponto 1: $P_j = (radius \cos(j \cdot \Delta\theta), 0, radius \sin(j \cdot \Delta\theta))$
- Ponto 2: $P_{j+1} = (radius \cos((j+1) \cdot \Delta\theta), 0, radius \sin((j+1) \cdot \Delta\theta))$

Triângulo: $[C, P_j, P_{j+1}]$

Características: - Base fechada com triângulation radial - Raio decresce linearmente com a altura - Topo converge em ponto singular a $(0, height, 0)$

4 Generator

4.1 Descrição

O Generator é uma aplicação de linha de comando que gera primitivas geométricas 3D e as armazena em ficheiros com extensão .3d. Funciona de forma totalmente independente do Engine, permitindo pré-computar todos os modelos necessários antes da renderização.

4.2 Funcionalidades

- **Geração de Primitivas**: Cria ficheiros 3D contendo definições de:
  - Planos (XZ, centrados na origem)
  - Caixas (todas as 6 faces)
  - Esferas (coordenadas esféricas)
  - Cones (base circular + lados convergentes)
- **Sistema de Pastas**: Cria automaticamente a pasta files3d/ se não existir
- **Formato XML**: Armazena geometria em formato XML estruturado para fácil parsing
- **Controle de Detalhe**: Permite especificar o número de subdivisões para cada primitiva

4.3 Exemplos de Utilização

```
# Gerar um plano com 1 unidade de comprimento e 3 divisões
generator plane 1 3 plane.3d

# Gerar uma caixa com 2 unidades de aresta e 3x3 subdivisões
generator box 2 3 box.3d

# Gerar uma esfera com raio 1, 10 fatias e 10 camadas
generator sphere 1 10 10 sphere.3d

# Gerar um cone com raio 1, altura 2, 4 fatias e 3 camadas
generator cone 1 2 4 3 cone.3d
```

5 Engine

5.1 Descrição

O Engine é a aplicação responsável por renderizar cenas 3D. Recebe como en- trada um ficheiro XML de configuração que especifica: - Dimensões da janela - Posição e orientação da câmara - Lista de ficheiros de modelos (.3d) a carregar e renderizar

5.2 Funcionalidades

Renderização: - Carregamento de múltiplos modelos 3D - Renderização via OpenGL com suporte a depth testing - Modo wireframe para visualizar estrutura dos modelos - Visualização de eixos de coordenadas (XYZ)

Interatividade: - Controles de teclado para alternar funcionalidades - Tratamento de janelas (redimensionamento, etc.) - Callbacks para eventos de entrada

Parsing de Configuração: - Leitura de ficheiros XML com tinyxml2 - Ex- tração de parâmetros da câmara (position, lookAt, up, projection) - Extração da lista de modelos a carregar

5.3 Fluxo de Execução

1. Inicialização:
   - Verificar argumentos de linha de comando (ficheiro XML obrigatório)
   - Criar câmara com valores padrão
   - Parsear ficheiro XML de configuração

2.Carregamento de Modelos:

- Para cada modelo listado no XML:
  - Abrir ficheiro .3d
  - Fazer parsing dos vértices dos triângulos
  - Armazenar em estrutura ModelData em memória

3.Setup OpenGL:

- Inicializar GLUT
- Criar janela com dimensões especificadas
- Registar callbacks de renderização

4.Renderização (Loop Principal):

- renderScene(): Desenha todos os modelos carregados
- changeSize(): Ajusta viewport em caso de redimensionamento
- processKeys(): Trata eventos de teclado
- processSpecialKeys(): Trata teclas especiais (setas, etc.)

5. Pipeline de Desemho:

- Depth testing ativado para oclusão correta
- Double buffering para animação suave
- Modo perspectiva com parâmetros da câmara

6 Formato de Ficheiros XML

Elementos Obrigatórios: - <world>: Raiz do documento - <window>: Especifica dimensões da janela - <camera>: Configuração da câmara - <group>: Grupo de modelos

Elementos Opcionais: - <up>: Vetor up da câmara (padrão: (0, 1, 0)) -<projection>: Parâmetros de projeção (padrão: fov=60, near=1, far=1000)

7 Conclusão

Este projeto estabeleceu as bases arquitectónicas para um motor de cenas 3D funcional. A separação clara entre **Generator** (produtor de geometria) e **Engine** (consumidor de geometria) permite uma arquitetura modular e escalável.

Principais Conquistas: - Arquitetura clara e extensível com duas aplicações complementares - Suporte a 4 primitivas geométricas com controle de detalhe
- Sistema de parsing XML robusto com tinyxml2 - Renderização OpenGL funcional com câmara configurável - Interface interativa para visualização de modelos

Fundações para Fases Futuras: - A estrutura modular permite adicionar fases subsequentes (transformações, iluminação, texturas) - O sistema XML é extensível para incluir novos parâmetros (cores, normais, etc.) - A câmara e modelo de dados estão prontos para suportar animações

O código está bem documentado, estruturado e pronto para expansão nas próximas fases do projeto.