Relatório de Computação Gráfica - Fase 1

Projeto CG_g16: Motor de Cenas 3D

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, cone, e adicionalmente o cilindro. 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.

---

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 extensão `.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, cone, cilindro) 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 CMakeCache.txt presente).

---

3 Primitivas Geométricas

3.1 Plano

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)

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

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: - `alphainc = 2 / slices` (incremento azimutal) - `betainc = / stacks` (incremento polar)

2. Para cada fatia e cada camada, gera dois triângulos conectando pontos consecutivos:

Pontos gerados:

```
P1: (radius·cos( )·cos( ), radius·sin( ), radius·cos( )·sin( ))
P2: (radius·cos( )·cos( + inc), radius·sin( ), radius·cos( )·sin( + inc))
P3: (radius·cos( + inc)·cos( ), radius·sin( + inc), radius·cos( + inc)·sin( ))
P4: (radius·cos( + inc)·cos( + inc), radius·sin( + inc), radius·cos( + inc)·sin( + inc))
```

onde varia de 0 a (eixo Y) e varia de 0 a 2 (no plano XZ).

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

O cone é uma primitiva que combina uma base circular 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) - `stacks`: Número de camadas verticais nos lados

Algoritmo: 1. Calcula parâmetros: - `alfa = 2 / slices` (ângulo entre fatias) - `alturaStack = height / stacks` (altura de cada camada) - `raioDecrement = radius / stacks` (redução de raio por camada)

2. Para cada fatia:
   - Gera triângulos da base (conectando ao centro)
   - Gera triângulos dos lados (conectando camadas consecutivas em coordenadas polares)

Coordenadas da base (para fatia i):

```
Centro: (0, 0, 0)
Ponto1: (radius·cos(i·alfa), 0, radius·sin(i·alfa))
Ponto2: (radius·cos((i+1)·alfa), 0, radius·sin((i+1)·alfa))
```

Coordenadas dos lados (para camada j da fatia i):

```
Raio_j = radius·(1 - j/stacks)
Altura_j = j·(height/stacks)
```

3.5 Cilindro

O cilindro combina duas bases circulares com lados verticais, mantendo o mesmo raio ao longo da altura.

Parâmetros: - `radius`: Raio do cilindro - `height`: Altura do cilindro - `slices`: Número de divisões em torno do eixo (lados) - `stacks`: Número de camadas verticais

Algoritmo: Semelhante ao cone, mas com as seguintes diferenças: - O raio permanece constante em todas as alturas - `raioDecrement = 0` (não há redução de raio) - Altura dividida entre stacks: `alturaStack = height / stacks`

---

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)
  - Cilindros (bases circulares + lados verticais)
- **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
```

```
# Gerar um cilindro com raio 1, altura 2, 10 fatias e 5 camadas
generator cylinder 1 2 10 5 cylinder.3d
```

Formato do Ficheiro de Saída:

Os ficheiros `.3d` possuem a seguinte estrutura XML:

```
<plane>
  <triangle>
    <vertex x="..." y="..." z="..." />
    <vertex x="..." y="..." z="..." />
    <vertex x="..." y="..." z="..." />
  </triangle>
  <!-- mais triângulos -->
</plane>
```

---

5 Engine

5.1 Descrição

O Engine é a aplicação responsável por renderizar cenas 3D. Recebe como entrada 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 - Extração 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

Estruturas de Dados Utilizadas:

```
struct Vertex {
    float x, y, z;
};

struct Face {
    int v1, v2, v3;  // Índices de vértices
};

struct ModelData {
    string filename;
    vector<Vertex> vertices;
    vector<Face> faces;
    bool loaded;
};
```

---

6 Formato de Ficheiros XML

6.1 Ficheiro de Configuração (Engine)

O Engine espera um ficheiro XML com a seguinte estrutura:

```
<world>
    <window width="800" height="600" />
    <camera>
        <position x="5" y="3" z="5" />
        <lookAt x="0" y="0" z="0" />
        <up x="0" y="1" z="0" />
        <projection fov="60" near="1" far="1000" />
    </camera>
```

```
    <group>
        <models>
            <model file="../generator/files3d/plane.3d" />
            <model file="../generator/files3d/cone.3d" />
        </models>
    </group>
</world>
```

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 5 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.

Data: 24 de Fevereiro de 2026
Projeto: CG_g16 - Motor de Cenas 3D - Fase 1
Linguagem: C++
Bibliotecas: OpenGL, GLUT, tinyxml2