Visualizador de Árvores Arteriais 2D

Trabalho de Computação Gráfica

Vinícius Nunes dos Anjos

Informações do Projeto

Disciplina:	Computação Gráfica
Tecnologias:	C++, OpenGL, GLFW, GLAD
Arquivos:	**main.cpp**, **TreeRenderer.cpp/h**, **VTKLoader.cpp/h**
Compilação:	C++17, OpenGL 3.3+

16 de dezembro de 2025

Sumário

1 Introdução

Este relatório descreve o desenvolvimento de um visualizador interativo de árvores arteriais bidimensionais, implementado em C++ utilizando as bibliotecas OpenGL, GLFW e GLAD. O sistema tem como objetivo principal fornecer uma ferramenta para visualização e análise de estruturas vasculares, permitindo carregar dados reais de arquivos VTK ou gerar árvores procedurais para fins de teste.

2 Arquitetura do Sistema

O sistema é composto por três componentes principais:

Componentes do Sistema

1. **main.cpp**: Ponto de entrada, gerenciamento de janela, entrada do usuário
2. **TreeRenderer.cpp/h**: Renderização OpenGL, shaders, transformações
3. **VTKLoader.cpp/h**: Carregamento e processamento de arquivos VTK

Figura 1: Arquitetura modular do sistema

3 Detalhamento dos Componentes

3.1 Main.cpp - Núcleo da Aplicação

O arquivo **main.cpp** é responsável pela coordenação geral da aplicação:

3.1.1 Inicialização

1. Inicialização do GLFW e criação da janela
2. Carregamento do GLAD para acesso às funções OpenGL
3. Configuração dos callbacks de entrada
4. Carregamento dos arquivos VTK disponíveis

3.1.2 Loop Principal

Algorithm 1 Loop principal de renderização

```
inicializarGLFW()
inicializarGLAD()
configurarCallbacks()
carregarArquivosVTK()
while !glfwWindowShouldClose() do
    calcularDeltaTime()
    processarEntradas()
    atualizarCamera(deltaTime)
    limparBuffer()
    aplicarTransformacoes()
    renderizarArvore()
    glfwSwapBuffers()
    glfwPollEvents()
end while
limparRecursos()
terminarGLFW()
```

3.1.3 Controles e Navegação

O sistema implementa um conjunto abrangente de controles:

Tecla	Função
ESC	Sair do programa
R	Resetar visualização
WASD	Movimentação suave
Clique + Arrastar	Movimentação com mouse
Q/E	Rotação suave
Scroll Mouse	Zoom suave
L	Alternar espessura adaptativa
C	Ciclar modos de cor
SETAS	Navegar entre árvores
I	Mostrar informações

Tabela 1: Controles da aplicação

3.2 TreeRenderer - Renderização OpenGL

3.2.1 Inicialização dos Shaders

```
#version 330 core
layout (location = 0) in vec2 aPos;
layout (location = 1) in vec3 aColor;
uniform mat4 transform;
out vec3 fragColor;
```

```
6
7  void main() {
8      gl_Position = transform * vec4(aPos, 0.0, 1.0);
9      fragColor = aColor;
10 }
```

Listing 1: Shaders do sistema

3.2.2 Análise da Estrutura da Árvore

O renderizador implementa algoritmos para análise estrutural:

```
1  int TreeRenderer::findRootSegment(const vector<Segment>& segments)
       {
2      vector<bool> hasParent(segments.size(), false);
3
4      for (size_t i = 0; i < segments.size(); i++) {
5          for (size_t j = 0; j < segments.size(); j++) {
6              if (i == j) continue;
7              float dist = abs(segments[j].end.x -
                   segments[i].start.x) +
8                           abs(segments[j].end.y -
                               segments[i].start.y);
9              if (dist < 0.001f) {
10                 hasParent[i] = true;
11                 break;
12             }
13         }
14     }
15
16     for (size_t i = 0; i < segments.size(); i++) {
17         if (!hasParent[i]) return static_cast<int>(i);
18     }
19     return 0;
20 }
```

Listing 2: Análise da árvore

3.2.3 Modos de Renderização

O sistema suporta quatro modos de coloração:

1. **Branco**: Renderização padrão
2. **Monocromático**: Verde uniforme
3. **Gradiente por Profundidade**: Violeta (folhas) → Vermelho (raiz)

4. **Gradiente por Descendentes**: Azul → Vermelho

```
if (useMonochrome) {
    r = 0.0f; g = 1.0f; b = 0.0f;
} else if (gradientMode) {
    r = 1.0f - normalizedDepth * 0.5f;
    g = 0.0f;
    b = normalizedDepth * 0.5f;
} else if (descendantsColorMode) {
    r = sqrt(normalizedDescendants);
    g = 0.0f;
    b = 1.0f - normalizedDescendants * normalizedDescendants;
} else {
    r = g = b = 1.0f;
}
```

Listing 3: Cálculo de cores

3.3 VTKLoader - Processamento de Dados

3.3.1 Processamento do Arquivo

Algorithm 2 Processamento de arquivo VTK

```
abrirArquivo(filename)
if !sucesso then
    return false
end if
lerCabecalho()
lerPontos()
lerConexoes()
lerRaios()
normalizarCoordenadas()
criarSegmentos()
return true
```

3.3.2 Normalização

Os dados são normalizados para visualização adequada:

$$x_{norm} = (x - centerX) \times scale$$

$$y_{norm} = (y - centerY) \times scale$$

onde $scale = \min\left(\frac{2}{maxX - minX}, \frac{2}{maxY - minY}\right) \times 0.8$

4 Algoritmos Implementados

4.1 Interpolação Suave

```
void updateSmoothTransform(float deltaTime) {
    auto lerp = [](float current, float target, float factor,
       float deltaTime) {
        return current + (target - current) * factor * deltaTime;
    };

    camera.translation[0] = lerp(camera.translation[0],
       camera.targetTranslation[0],
                                 config.smoothFactor, deltaTime);
    camera.translation[1] = lerp(camera.translation[1],
       camera.targetTranslation[1],
                                 config.smoothFactor, deltaTime);
    camera.rotation = lerp(camera.rotation, camera.targetRotation,
                           config.smoothFactor, deltaTime);
    camera.scale = lerp(camera.scale, camera.targetScale,
                        config.smoothFactor, deltaTime);
}
```

Listing 4: Interpolação linear da câmera

4.2 Cálculo de Matriz de Transformação

A matriz de transformação 4x4 é calculada como:

$$M = \begin{bmatrix} \cos\theta \cdot s & -\sin\theta \cdot s & 0 & t_x \\ \sin\theta \cdot s & \cos\theta \cdot s & 0 & t_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

onde:

- θ: ângulo de rotação
- s: escala
- t_x, t_y: translação

```
void updateTransformMatrix() {
    fill_n(transformMatrix, 16, 0.0f);
    transformMatrix[0] = transformMatrix[5] = transformMatrix[10]
       = transformMatrix[15] = 1.0f;

```

```
    float cosR = cos(camera.rotation);
    float sinR = sin(camera.rotation);
    transformMatrix[0] = cosR * camera.scale;
    transformMatrix[1] = -sinR * camera.scale;
    transformMatrix[4] = sinR * camera.scale;
    transformMatrix[5] = cosR * camera.scale;

    transformMatrix[12] = camera.translation[0];
    transformMatrix[13] = camera.translation[1];
}
```

Listing 5: Cálculo da matriz

4.3 Busca em Largura (BFS) para Profundidade

Algorithm 3 Cálculo de profundidade usando BFS

Require: Lista de adjacência *children*, índice *root*
Ensure: Vetor *depth* com profundidades

```
depth[root] ← 0
Q ← fila vazia
Q.push(root)
while Q ≠ ∅ do
    current ← Q.front()
    Q.pop()
    for child ∈ children[current] do
        if depth[child] = −1 then
            depth[child] ← depth[current] + 1
            Q.push(child)
        end if
    end for
end while
```

4.4 Contagem de Descendentes (DFS)

Algorithm 4 Cálculo recursivo de descendentes

```
calculateDescendantsnode
count ← 0
for child ∈ children[node] do
    count ← count + 1 + calculateDescendants(child)
end for
descendantCount[node] ← count
return count
```

5 Modos de Visualização

5.1 Espessura Adaptativa

A espessura das linhas pode ser fixa ou adaptativa:

$$\text{thickness} = \begin{cases} \text{lineWidth} & \text{se thicknessMode} = \text{false} \\ 2.0 + \text{normalizedDescendants} \times 13.0 & \text{se thicknessMode} = \text{true} \end{cases}$$

5.2 Sistema de Cores

Modo	Fórmula R	Fórmula G	Fórmula B
Branco	1.0	1.0	1.0
Monocromático	0.0	1.0	0.0
Profundidade	$1 - 0.5d$	0.0	$0.5d$
Descendentes	$\sqrt{d}$	0.0	$1 - d^2$

Tabela 2: Fórmulas de coloração (d: valor normalizado)

6 Arquitetura de Renderização

6.1 Buffer de Vértices

Os dados são organizados em formato intercalado (interleaved):

```
[x1, y1, r1, g1, b1, x2, y2, r2, g2, b2, ...]
```

```
RenderData TreeRenderer::prepareRenderData(const vector<Segment>&
    segments) {
    RenderData data;
    data.vertices.reserve(segments.size() * 4);
    data.colors.reserve(segments.size() * 6);
    data.thicknesses.reserve(segments.size());

    for (size_t i = 0; i < segments.size(); i++) {
        data.vertices.push_back(segment.start.x);
        data.vertices.push_back(segment.start.y);
        data.colors.push_back(r);
        data.colors.push_back(g);
        data.colors.push_back(b);

        data.vertices.push_back(segment.end.x);
```

```
        data.vertices.push_back(segment.end.y);
        data.colors.push_back(r);
        data.colors.push_back(g);
        data.colors.push_back(b);

        data.thicknesses.push_back(thickness);
    }
    return data;
}
```

Listing 6: Preparação de dados de renderização

6.2 Renderização por Segmento

Para o modo de espessura adaptativa, cada segmento é renderizado individualmente:

```
if (thicknessMode && !data.thicknesses.empty()) {
    for (size_t i = 0; i < segments.size(); i++) {
        float thickness = clamp(data.thicknesses[i], 1.0f, 10.0f);
        glLineWidth(thickness);

        vector<float> segmentData;
        // ... preparar dados do segmento i
        glBufferData(GL_ARRAY_BUFFER, segmentData.size() *
           sizeof(float),
                     segmentData.data(), GL_STATIC_DRAW);
        glDrawArrays(GL_LINES, 0, 2);
    }
}
```

Listing 7: Renderização individual por segmento

7 Interface e Interação

7.1 Sistema de Câmera

A câmera implementa transformações em 2D:

- **Translação**: Movimento nos eixos X e Y
- **Rotação**: Rotação em torno do centro
- **Escala**: Zoom in/out
- **Interpolação**: Movimento suave entre estados

Parâmetro	Valor Padrão	Descrição
moveSpeed	0.0005	Velocidade de movimento (WASD)
rotationSpeed	0.0005	Velocidade de rotação (Q/E)
zoomSpeed	0.4	Velocidade de zoom (scroll)
smoothFactor	3.0	Fator de suavização
dragSensitivity	0.0013	Sensibilidade do arrasto
minScale	0.1	Zoom mínimo
maxScale	5.0	Zoom máximo
translationLimit	2.0	Limite de translação

Tabela 3: Parâmetros de configuração da câmera

7.2 Processamento de Entrada

O sistema usa callbacks do GLFW para entrada:

```
glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);
glfwSetKeyCallback(window, key_callback);
glfwSetScrollCallback(window, scroll_callback);
glfwSetMouseButtonCallback(window, mouse_button_callback);
glfwSetCursorPosCallback(window, cursor_position_callback);
```

Listing 8: Callbacks GLFW

8 Carregamento de Dados

8.1 Busca de Arquivos

O sistema procura arquivos VTK em múltiplos diretórios:

```
void loadTreeFiles() {
    vector<string> folders = {"data/Nterm_064", "data/Nterm_128",
       "data/Nterm_256"};

    for (const auto& folder : folders) {
        if (!fs::exists(folder) || !fs::is_directory(folder))
           continue;

        for (const auto& entry : fs::directory_iterator(folder)) {
            if (!entry.is_regular_file()) continue;
            string path = entry.path().string();
            if (entry.path().extension() != ".vtk") continue;

            treeFiles.push_back(path);
            treeFileNames.push_back(createFriendlyName(path));
        }
    }
```

```
16
17      if (treeFiles.empty()) {
18          // Fallback para arquivos padr o
19          treeFiles = {
20              "data/Nterm_064/tree2D_Nterm0064_step0064.vtk",
21              "data/Nterm_064/tree2D_Nterm0064_step0008.vtk",
22              "data/Nterm_128/tree2D_Nterm0128_step0128.vtk",
23              "data/Nterm_256/tree2D_Nterm0256_step0256.vtk"
24          };
25      }
26  }
```

Listing 9: Carregamento de arquivos VTK

8.2 Geração Procedural

Se nenhum arquivo VTK for encontrado, o sistema gera uma árvore procedural:

Algorithm 5 Geração de árvore procedural

generateBranch$start$, $direction$, $length$, $radius$, $depth$, $parent$
if $depth \leq 0$ OR $length < 0.01$ **then**
  return -1
end if
$end \leftarrow start + direction \times length + noise$
$criarSegmento(start, end, radius)$
if $depth > 1$ **then**
  $numBranches \leftarrow (depth > 3)?2 : 1$
  for $i = 0$ TO $numBranches - 1$ **do**
    $angle \leftarrow (i = 0)?0.5 : -0.5$
    $newDir \leftarrow rotate(direction, angle)$
    $generateBranch(end, newDir, length \times 0.6,$
      $radius \times 0.7, depth - 1, endIndex)$
  end for
end if

9 Conclusão

9.1 Conquistas do Projeto

O visualizador implementado alcançou os seguintes objetivos:

1. **Carregamento de Dados**: Suporte a arquivos VTK reais com fallback procedural
2. **Visualização Avançada**: Múltiplos modos de cor e espessura

3. **Interação Fluida**: Sistema de câmera suave e responsivo
4. **Análise Estrutural**: Algoritmos para extração de métricas

9.2 Considerações Finais

O Visualizador de Árvores Arteriais 2D representa uma ferramenta completa para análise e visualização de estruturas vasculares. A implementação demonstra conceitos fundamentais de computação gráfica, incluindo:

- **Transformações Geométricas**: Matrizes de transformação, interpolação
- **Renderização OpenGL**: Shaders, VAO/VBO, estados de renderização
- **Processamento de Dados**: Análise de grafos, normalização
- **Interação**: Callbacks, entrada de mouse/teclado

10 Extras

Requisitos do Sistema Utilizados

- **Sistema Operacional**: Windows
- **Compilador**: C++17 (g++ 7+, Makefile)
- **Bibliotecas**: GLFW3, GLAD, OpenGL 3.3+

Comandos de Compilação

```
# Para somente compilar
make
# Compilar e rodar
make run
# Obs: Garantir que glfw3.dll esteja no diretório raiz
# Somente execução
./programa
```

Github

`https://github.com/Vianjus/TP_Comp_Grafica.git`

Créditos

Através do site e dos vídeos no Youtube do TerminalRoot consegui fazer esse trabalho. https://terminalroot.com.br/2024/02/tutorial-de-opengl-para-iniciantes.html