UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Trabalho Prático: Simulação de Bando (Boids) 3D

Autores:
Mateus Ryan de Castro Lima - 2021036752
Rainer Menezes Vieira Silva - 2017068785

Professor:
Renato Ferreira

Belo Horizonte
Novembro de 2025

Sumário

1 Introdução

Este trabalho apresenta a implementação de uma simulação 3D baseada no modelo de comportamento coletivo *Boids*, proposto por Craig Reynolds. O objetivo é reproduzir o movimento de um bando de entidades autônomas (boids), obedecendo a três regras locais:

- **Separação:** evitar colisões com vizinhos;
- **Coesão:** aproximar-se do centro de massa do bando;
- **Alinhamento:** alinhar a direção com os vizinhos.

Além disso, foi implementado um boid especial (boid-líder), cujo movimento influencia o bando e pode ser controlado pelo usuário em tempo real.

A simulação foi desenvolvida em C++, utilizando OpenGL (perfil de compatibilidade), GLM, GLFW e GLAD, com gerenciamento do build via `Makefile`.

2 Ambiente de Desenvolvimento

O projeto foi implementado utilizando:

- **C++17** como linguagem de programação;
- **g++** (MinGW 64 bits) como compilador;
- **GLFW** para criação de janela e captura de entrada;
- **GLAD** para carregamento das funções do OpenGL;
- **GLM** para operações vetoriais e matriciais;
- **GLU** para funções auxiliares típicas do modo compatível.

Os diretórios do projeto estão organizados da seguinte forma:

- `src/`: código-fonte principal;
- `include/`: arquivos de cabeçalho;
- `lib/`: bibliotecas utilizadas no projeto;
- `obj/`: objetos compilados;
- `Makefile`: script de automação de compilação.

3 Construção do Mundo 3D

A cena tridimensional consiste em três elementos principais: o chão (grid), a torre central e os boids animados.

3.1 Grid do Chão

O chão é representado por um grid de linhas paralelas aos eixos X e Z. Esse grid auxilia na avaliação visual da profundidade e posição relativa dos boids.

Seu desenho é feito utilizando:

- laços de repetição que varrem intervalos simétricos;
- chamadas a `glBegin(GL_LINES)` e `glVertex3f(...)`.

3.2 Torre Central

A torre, localizada no centro da cena, é uma estrutura cônica construída com:

- `gluCylinder` para gerar a superfície do cone;
- transformações `glTranslatef` e `glRotatef` para posicionamento;
- material próprio para refletir iluminação corretamente.

3.3 Iluminação

A simulação utiliza iluminação tradicional do OpenGL, com:

- luz ambiente, difusa e especular;
- posicionamento da luz acima da cena;
- materiais definidos para boids e objetos estáticos.

4 Arquitetura da Simulação

A simulação foi dividida em três componentes principais:

- **Boid**: responsável pela física e regras comportamentais;
- **Renderer**: responsável pela renderização da cena;
- **Controle/Entrada**: responsável pela interação do usuário.

4.1 Cálculo do Centro do Bando

O ponto médio do bando é calculado em cada quadro:

$$centro = \frac{1}{N} \sum_{i=1}^{N} pos_i$$

Esse valor é utilizado principalmente:

- na lógica da câmera (apontando sempre para o centro);
- como referência de coesão do bando.

5 Regras Comportamentais

Cada boid atualiza sua aceleração conforme as forças resultantes:

```
void Boid::calculateForces(const std::vector<Boid>& flock,
                           const glm::vec3& goalPosition) {
    acceleration = glm::vec3(0.0f);

    glm::vec3 sep = separate(flock);
    glm::vec3 ali = align(flock);
    glm::vec3 coh = cohere(flock);
    glm::vec3 bounds = checkBounds();
    glm::vec3 seek = steerTo(goalPosition);

    applyForce(sep * PESO_SEPARACAO);
    applyForce(ali * PESO_ALINHAMENTO);
    applyForce(coh * PESO_COESAO);
    applyForce(bounds * 1.5f);
    applyForce(seek * PESO_SEEK_GOAL);
}
```

Listing 1: Trecho da função Boid::calculateForces

6 Uso de GLM e GLU

As bibliotecas de matemática e utilidades foram essenciais:

- **GLM** foi utilizado para representar vetores e posições com `glm::vec3`, calcular distâncias/direções (`length`, `normalize`) e definir projeções com `glm::perspective`.
- **GLU** foi utilizado para funções de conveniência do modo compatível, como `gluLookAt` para controlar a câmera e `gluCylinder` para desenhar a torre.

7 Câmeras e Interação

A simulação possui três modos de câmera:

1. Vista do topo da torre;
2. Vista atrás do bando, seguindo sua direção de movimento;
3. Vista lateral, perpendicular ao vetor velocidade.

A interação do usuário é feita pelo teclado:

- `I,J,K,L,U,O`: movem o boid-líder;
- `+` e `-`: adicionam e removem boids;
- `C`: alterna os modos de câmera.

8 Calibração dos Parâmetros

Os pesos das forças influenciam fortemente o comportamento:

- separação alta → bando mais disperso;
- coesão alta → bando mais compacto;
- alinhamento alto → movimento mais suave;
- seek alto → forte atração ao boid-líder.

Vários testes foram realizados para obter estabilidade e naturalidade.

9 Funcionalidades Extras Implementadas

Além das funcionalidades básicas obrigatórias, foram implementadas quatro funcionalidades extras, totalizando 20% de pontos adicionais. Esta seção descreve os fundamentos teóricos, desafios técnicos e estratégias de implementação de cada uma.

9.1 Sombras por Projeção Planar (5%)

9.1.1 Fundamentação Teórica

A renderização de sombras contribui significativamente para a percepção de profundidade e posicionamento espacial dos objetos em cenas tridimensionais. Neste trabalho, foi implementada a técnica de *projeção planar*, que calcula uma matriz de transformação capaz de "achatar" objetos sobre um plano (o chão), criando uma silhueta escurecida.

A matriz de projeção planar $\mathbf{M}_{sombra}$ é derivada da posição da luz direcional $\mathbf{L} = (L_x, L_y, L_z)$ e da normal do plano $\mathbf{N} = (0, 1, 0)$:

$$\mathbf{M}_{sombra} = \mathbf{I} - \frac{\mathbf{L} \otimes \mathbf{N}}{\mathbf{L} \cdot \mathbf{N}}$$

onde $\otimes$ denota o produto tensorial (outer product) e $\mathbf{I}$ é a matriz identidade 4×4.

9.1.2 Implementação

A implementação foi dividida em três componentes principais:

1. **Classe Shadow**: Responsável pelo cálculo da matriz de projeção e gerenciamento do estado OpenGL para renderização de sombras;
2. **Método getShadowMatrix()**: Calcula a matriz 4×4 de projeção planar utilizando a direção normalizada da luz;
3. **Renderização em duas passadas**: Primeiro renderizam-se as sombras (com iluminação desabilitada e *blending* ativo), depois os objetos normais.

O código a seguir ilustra o cálculo da matriz de sombra:

```
glm::mat4 Shadow::getShadowMatrix(const glm::vec3& lightDir,
                                  float planeY) {
    glm::vec3 l = glm::normalize(lightDir);
    glm::vec3 n(0.0f, 1.0f, 0.0f); // Normal do plano (chao)

    float dot = glm::dot(n, l);
    glm::mat4 shadowMat(1.0f);

    // Construcao da matriz 4x4 de projecao planar
    shadowMat[0][0] = dot - l.x * n.x;
    shadowMat[1][0] = -l.x * n.y;
    // ... demais elementos da matriz

    return shadowMat;
}
```

Listing 2: Cálculo da matriz de projeção planar

9.1.3 Desafios e Soluções

Z-fighting: A sobreposição das sombras com o plano do chão causava artefatos visuais (*flickering*). Solução: aplicar um pequeno deslocamento vertical ($y = 0.01$) às sombras.

Transparência: Para criar o efeito visual de sombra, utilizou-se *alpha blending* com `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)` e cor preta semi-transparente ($\alpha = 0.4$).

Desempenho: A renderização adicional de todos os boids como sombras poderia impactar a taxa de quadros. Utilizou-se `glDepthMask(GL_FALSE)` para evitar escritas desnecessárias no *depth buffer*.

9.2 Reshape: Redimensionamento Adaptativo (5%)

9.2.1 Fundamentação Teórica

O redimensionamento de janela é essencial para proporcionar flexibilidade ao usuário. Quando as dimensões da janela mudam, três parâmetros devem ser recalculados:

1. **Viewport**: Região da janela onde ocorre a renderização;
2. **Aspect Ratio**: Proporção largura/altura, que afeta a matriz de projeção;
3. **Matriz de Projeção Perspectiva**: Recalculada para evitar distorções.

A matriz de projeção perspectiva é dada por:

$$\mathbf{P} = perspective(fov, aspectRatio, z_{near}, z_{far})$$

onde `fov` é o campo de visão em radianos, e z_{near} e z_{far} definem o *frustum* de visualização.

9.2.2 Implementação

O GLFW oferece o mecanismo de *callbacks* para notificar a aplicação quando o *framebuffer* é redimensionado. A implementação consistiu em:

1. Registrar o callback `glfwSetFramebufferSizeCallback()`;
2. Recalcular viewport com `glViewport(0, 0, width, height)`;
3. Recalcular a matriz de projeção utilizando o novo *aspect ratio*.

```
void framebufferSizeCallback(GLFWwindow* window,
                             int width, int height) {
    // Atualiza viewport e matriz de projecao
    setupProjection(window);
    std::cout << "Janela redimensionada: "
              << width << "x" << height << std::endl;
}
```

Listing 3: Callback de redimensionamento

Esta funcionalidade garante que a visualização 3D permaneça proporcional e sem distorções, independentemente das dimensões da janela.

9.3 Fog: Efeito de Profundidade Atmosférica (5%)

9.3.1 Fundamentação Teórica

O *fog* (névoa) é uma técnica de renderização que simula a atenuação da visibilidade com a distância, aumentando o realismo e a percepção de profundidade. Matematicamente, a cor final de um fragmento é interpolada entre sua cor original e a cor da névoa:

$$C_{final} = C_{objeto} \cdot f + C_{fog} \cdot (1 - f)$$

onde f é o fator de névoa, calculado de acordo com o modo escolhido. No modo linear:

$$f = \frac{z_{end} - z}{z_{end} - z_{start}}$$

sendo z a distância do fragmento à câmera, e z_{start} e z_{end} os limites de início e fim do efeito.

9.3.2 Implementação

A implementação utilizou as funções nativas do OpenGL para fog:

```
void setupFog() {
    GLfloat fogColor[] = {0.3f, 0.3f, 0.5f, 1.0f};

    glFogi(GL_FOG_MODE, GL_LINEAR);
    glFogfv(GL_FOG_COLOR, fogColor);
    glFogf(GL_FOG_START, 15.0f);
```

```
7       glFogf(GL_FOG_END, 60.0f);
8       glHint(GL_FOG_HINT, GL_NICEST);
9   }
```

Listing 4: Configuração do fog linear

O efeito pode ser ativado/desativado dinamicamente através da tecla `F`, utilizando `glEnable(GL_FOG)` e `glDisable(GL_FOG)`.

9.3.3 Ajustes de Parâmetros

A escolha dos valores $z_{start} = 15.0$ e $z_{end} = 60.0$ foi feita empiricamente, considerando o tamanho do mundo simulado (100×100 unidades). Objetos mais próximos que 15 unidades permanecem nítidos, enquanto objetos além de 60 unidades ficam completamente envolvidos pela névoa.

A cor da névoa foi definida como azul-acinzentada $(0.3, 0.3, 0.5)$ para harmonizar com a cor de fundo da cena, criando uma transição visual suave.

9.4 Modo de Pausa e Debug (5%)

9.4.1 Fundamentação Teórica

Simulações físicas em tempo real dependem de um *game loop* que atualiza o estado do sistema a cada quadro. A capacidade de pausar e inspecionar o estado interno é fundamental para:

- Validação do comportamento das regras de Boids;
- Análise de estabilidade numérica;
- Depuração de artefatos visuais;
- Apresentação didática do funcionamento interno.

9.4.2 Implementação

Foram implementados três modos de operação:

1. **RUNNING**: Simulação executando normalmente, com `flock.update(deltaTime)` sendo chamado a cada quadro;
2. **PAUSED**: Simulação congelada. O método `update()` não é chamado, mas a renderização continua ativa. Boids podem ser adicionados ou removidos mesmo neste estado;
3. **DEBUG_STEP**: Modo de execução passo-a-passo. A cada pressionamento da tecla `SPACE`, um único quadro é simulado com Δt fixo de 0.016 segundos (≈ 60 FPS), e informações detalhadas são impressas no console.

```
// No loop principal
if (!isPaused && !debugMode) {
    flock.update(deltaTime);
}

// No modo debug, avanco manual
if (debugMode && teclaSpacePressionada) {
    flock.update(0.016f); // Timestep fixo
    printDebugInfo(flock, frameCount);
}
```

Listing 5: Controle do loop de simulação

9.4.3 Informações de Debug

A função `printDebugInfo()` exibe:

- Número total de boids;
- Posição e velocidade do primeiro boid (representativo);
- Velocidade escalar (magnitude do vetor);
- Centro de massa do bando;
- Posição do boid-objetivo.

Exemplo de saída:

```
DEBUG - Frame 42

Boids: 50
Boid[0] Pos: (12.34, 5.67, -8.90)
Boid[0] Speed: 3.45
Goal: (20.00, 10.00, 0.00)
Center: (15.23, 8.91, -2.34)
```

Esta funcionalidade foi particularmente útil para verificar a convergência do bando em direção ao objetivo e identificar situações de instabilidade numérica.

9.5 Integração das Funcionalidades Extras

Todas as funcionalidades extras foram projetadas para serem independentes e não invasivas, podendo ser ativadas/desativadas dinamicamente:

- `S`: Toggle de sombras;
- `F`: Toggle de fog;

- `P`: Toggle de pausa;
- `D`: Toggle de modo debug;
- `SPACE`: Avançar um frame (apenas em modo debug).

A arquitetura modular permitiu adicionar estas funcionalidades sem modificar substancialmente o código base da simulação, mantendo a separação de responsabilidades entre módulos de renderização, física e interface.

10 Resultados

A simulação produz um bando coeso, estável e com aparência natural. Aqui devem ser incluídas imagens reais da execução:

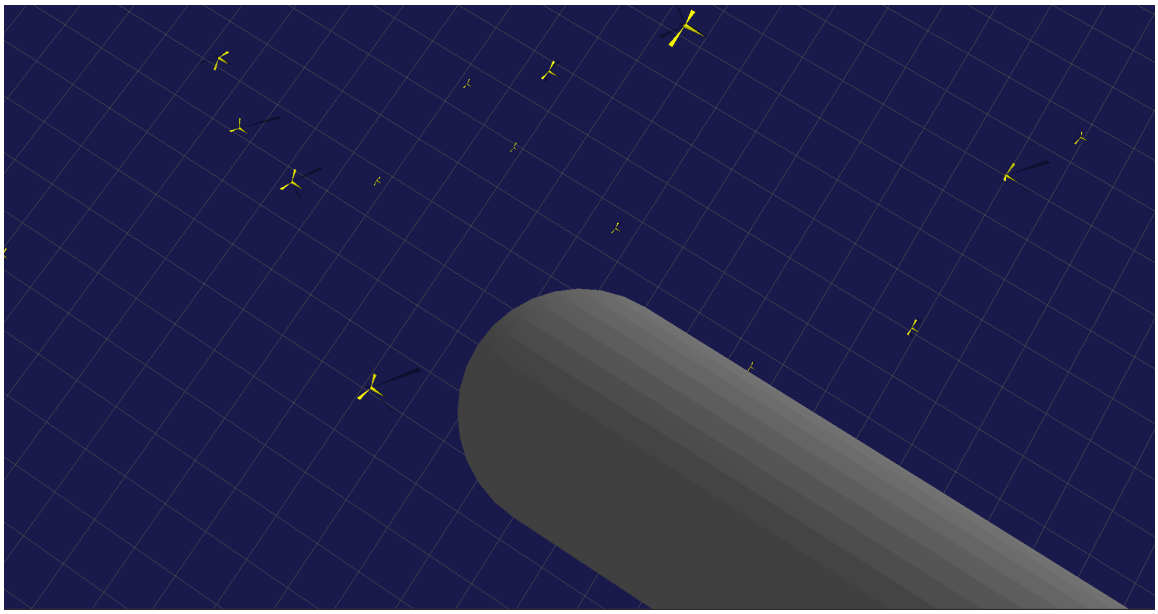

Figura 1: Câmera posicionada no topo da torre.

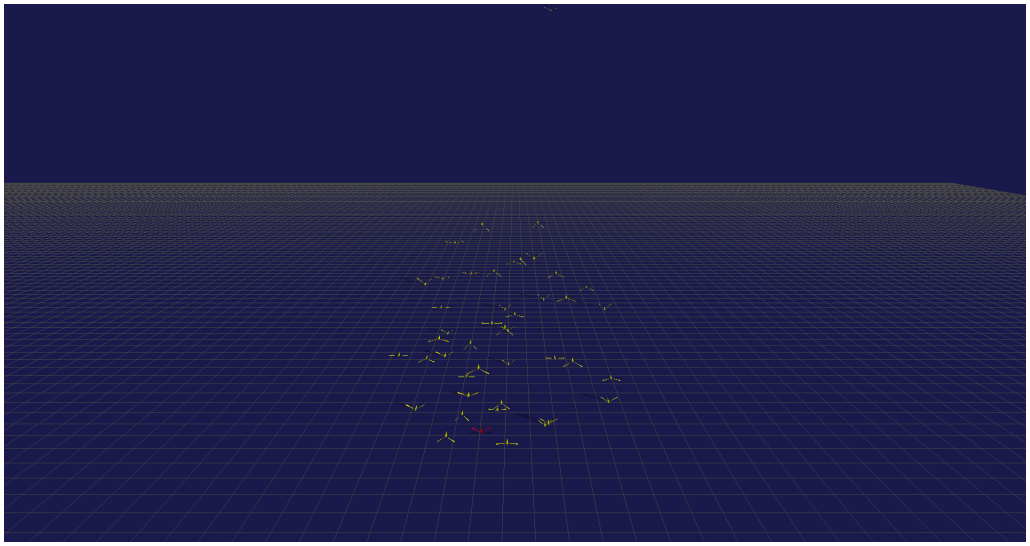

Figura 2: Bando seguindo o boid-objetivo controlado pelo usuário.

11 Conclusão

A simulação implementada cumpre todos os requisitos do trabalho, permitindo visualizar comportamento emergente de boids em ambiente tridimensional. O uso de bibliotecas modernas (GLM, GLFW, GLAD) aliado ao OpenGL tradicional resultou em um sistema funcional e interativo.

Referências

[1] Reynolds, Craig. *Flocks, Herds, and Schools: A Distributed Behavioral Model.* Computer Graphics, 1987.

[2] GLFW – OpenGL Framework. `https://www.glfw.org/`

[3] GLM – OpenGL Mathematics. `https://github.com/g-truc/glm`

[4] GLAD – OpenGL Loader. `https://glad.dav1d.de/`