**UNIVERSIDADE FEDERAL DE PELOTAS**
**CENTRO DE DESENVOLVIMENTO TECNOLÓGICO**


# Implementação de um algoritmo de Boids em C com visualização gráfica usando Python

Christian Kringel, Henrique Versiani, Pedro Ravazolo, Rodrigo Mendes, Vinicius Hallmann

**Conceitos de Linguagem de Programação - 2025/1**
**Professor: Dr. Gerson Geraldo Cavalheiro**

# 1. Introdução

Este projeto foi elaborado para a disciplina de Conceitos de Linguagens de Programação da Universidade Federal de Pelotas. A proposta dada consiste no desenvolvimento de uma aplicação gráfica utilizando duas linguagens de programação distintas, escolhidas livremente. O principal desafio estava na integração entre essas linguagens, sendo exigido que uma delas fosse responsável pela interface com o usuário e pela exibição gráfica, enquanto a outra cuidasse dos cálculos envolvidos na aplicação. Nesse contexto, o grupo optou por implementar o algoritmo ”Boids”, (originalmente proposto e desenvolvido por Craig Reynolds em 1986) um sistema de vida artificial que modela dinâmicas e comportamentos observados em de bandos de pássaros, cardumes de peixes ou outros animais que se movimentam em grupo.

**Boid** é uma abreviação de "bird-oid object" (objeto similar a um pássaro), o objetivo é demonstrar como comportamentos de grupo complexos e realistas podem emergir a partir de um conjunto simples de regras seguidas por agentes individuais. Cada "boid" na simulação opera de forma autônoma, reagindo apenas a seus vizinhos imediatos, sem ter conhecimento do estado global do bando.

Há 3 regras principais em que a simulação se baseia:

- **Separação**: Evita colisões com vizinhos próximos.
- **Alinhamento**: Ajusta as direções do boid com base nos vizinhos próximos.
- **Coesão**: Leva o boid em direção ao centro de massa do boid vizinho.

Além dessas regras, a simulação implementa funcionalidades adicionais, como limites de velocidade (mínima e máxima), barreiras nas bordas da tela para manter o bando contido e interações com o mouse, permitindo ao usuário atrair ou afastar os boids em tempo real entre outras funcionalidades

# 2. Arquitetura Geral do Projeto

Para implementar essa simulação, o projeto foi dividido em duas partes principais: Backend e Frontend. A lógica computacional do comportamento dos boids, que envolve cálculos vetoriais e regras de movimentação, foi inteiramente implementada em **linguagem C**, com foco em desempenho e paralelização utilizando OpenMP **(Backend)**.

A interface gráfica e o controle interativo foram desenvolvidos em **Python**, utilizando a biblioteca Pygame para renderizar os boids em tempo real **(Frontend)**. A comunicação entre as duas linguagens é feita por meio da biblioteca **ctypes**, que

permite que estruturas e funções escritas em C sejam utilizadas diretamente em Python.

## 2.1 Detalhamento da implementação

A **linguagem C** foi escolhida para ser a responsável pela parte central da simulação, ou seja, toda a lógica de cálculo intensivo e estruturas de dados fundamentais. A principal razão para essa escolha é o seu alto desempenho e controle de baixo nível sobre a memória, características cruciais para uma simulação que precisa atualizar a posição e velocidade de milhares de entidades dezenas de vezes por segundo.

Os **3 módulos principais** do Back-end em C são:

- **boids.c/h:** Criação, inicialização e destruição dos boids. Responsável pela gerência da alocação e liberação de memória e definição do estado inicial das entidades na simulação.
- **simulation.c/h:** Atualização das posições de estado de cada boid e aplicação das regras.
- **grid.c/h:** Acelera a busca por vizinhos usando divisão espacial (*spatial hashing*).

**Estruturas de dados fundamentais:** As principais estruturas de dados são definidas no arquivo **datatypes.h**

- **Position** e **Velocity** são structs que armazenam as coordenadas e as velocidades.
- **Entity** combina uma **Position** e uma **Velocity** para representar um único boid.
- **Boids** é uma struct que contém um array de todas as **Entity** e a contagem total.
- **Grid** é a struct usada para a otimização espacial, contendo informações sobre as células da grade.

A biblioteca OpenMP (#pragma omp parallel for) é utilizada para paralelizar os loops de cálculo mais pesados, distribuindo a carga de trabalho entre os núcleos da CPU e acelerando significativamente a simulação.
A linguagem **Python**, em conjunto com a biblioteca Pygame, foi utilizada para construir a interface com o usuário da aplicação. A Linguagem foi escolhida por sua simplicidade, legibilidade e pela vasta disponibilidade de bibliotecas para o desenvolvimento rápido de interfaces gráficas. Python foi utilizado para criar e gerenciar a janela gráfica da simulação, renderizar os boids na tela e gerenciar os eventos de entrada (como movimentação e clique do mouse).

Os **módulos principais** do Front-end em Python são:

- **main.py**: Ponto de entrada da aplicação. Inicializa a simulação carregando configurações iniciais e coordenação entre os diferentes módulos.
- **app.py**: Classe principal da aplicação que gerencia o loop principal da aplicação, que a cada iteração processa eventos de entrada, invoca a atualização da lógica da simulação (chamando a função em C) e atualiza a interface gráfica.
- **simulation.py**: Responsável por configurar e chamar as funções da biblioteca C compilada, transformando as estruturas Python em tipos compatíveis com ctypes. Gerencia a inicialização dos boids, chamadas de atualização da simulação e limpeza de memória.
- **renderer.py**: Sistema de renderização gráfica responsável por toda a apresentação visual. Ele inicia o Pygame, cria a janela e a cada quadro, desenha todos os boids na tela.
- **input_handler.py:** Captura e trata eventos de mouse (atração/repulsão de boids), teclado (controles de simulação), e outros dispositivos de entrada, convertendo-os em ações apropriadas na simulação.
- **UI.py**: Interface de usuário interativa que permite controle em tempo real dos parâmetros da simulação. Implementa elementos como botões, sliders, exibição de FPS e indicadores visuais que podem ser modificados dinamicamente.
- **globals.py** e **state.py**: Centralizam as variáveis de configuração (número de boids, velocidades, cores) e o estado global da aplicação (como referências à biblioteca carregada), facilitando ajustes e a manutenção do código.

# 3. Integração da interface: Biblioteca ctypes

A comunicação entre C e Python foi realizada utilizando a biblioteca nativa ctypes do Python. Esta biblioteca permite carregar bibliotecas compartilhadas (como arquivos .dll no Windows ou .so no Linux) em um processo Python e chamar funções contidas nela como se fossem funções nativas. A implementação seguiu um processo bem definido para garantir a integridade e o desempenho da comunicação:

1. **Compilação do Código C**: Primeiramente, todo o código-fonte do back-end em C é compilado não como um executável, mas como uma biblioteca de vínculo dinâmico (shared library). Isso gera um único arquivo que expõe as funções C (initialize_boids, update_boids, etc.) para que possam ser chamadas por outros programas.
2. **Espelhamento das Estruturas de Dados:** No arquivo c_definitions/c_structures.py, cada struct definida em C é recriada como uma classe Python que herda de ctypes.Structure. Este passo é crucial, pois instrui o Python a alocar memória para esses objetos de uma forma que seja bit a bit compatível com o layout de memória das structs em C.

3. **Carregamento da Biblioteca**: O módulo c_definitions/c_interfaces.py localiza o arquivo da biblioteca compartilhada e a carrega na memória do processo Python utilizando a função ctypes.CDLL(). A partir deste momento, as funções C estão acessíveis dentro do Python.
4. **Definição das Assinaturas de Funções**: Para evitar erros de interpretação de dados, as assinaturas de cada função C são explicitamente definidas no Python. Isso é feito configurando os atributos .argtypes (uma lista dos tipos de dados dos parâmetros de entrada) e .restype (o tipo de dado do valor de retorno). Por exemplo, ao definir que um argumento é um ctypes.POINTER(Boids), garantimos que o Python passará um ponteiro de memória válido, e não uma cópia do objeto.
5. **Execução e Troca de Dados**: Com a interface configurada, o Python pode chamar diretamente as funções C. A chamada simulation.update() em Python invoca a função update_boids em C, passando os ponteiros para as estruturas de dados dos boids. O C executa os cálculos intensivos e modifica os dados (posições e velocidades) diretamente nos blocos de memória apontados. Como Python e C estão compartilhando essa memória, as atualizações feitas pelo C são imediatamente visíveis para o Python, que então prossegue para a renderização do novo estado na tela.