GPU-Accelerated Boids Simulation with Naive and Grid-Based Neighborhood Evaluation

Jan Koča
kocajan@fel.cvut.cz

Final report for the course B4M39GPU
Winter semester 2025/2026

***Abstract*—This report presents the design, implementation, and evaluation of a GPU-accelerated boids simulation based on an extended flocking model. The simulation supports multiple agent types, including basic boids, predators, and static obstacles, as well as interactive user control, configurable boundary conditions, and both two- and three-dimensional domains. Four simulation variants are implemented: sequential naive, sequential grid-based, parallel naive (GPU), and parallel grid-based (GPU). All variants share an identical behavioral model and parameterization, differing only in neighborhood evaluation strategy and execution platform. A unified application framework enables both interactive visualization and automated experiment execution. Performance experiments compare execution time across implementations, scenarios, and hardware platforms, while state-recording experiments verify behavioral consistency. Results demonstrate that GPU-based implementations significantly outperform CPU-based ones, and that grid-based neighborhood search improves performance primarily in scenarios with uniform spatial distributions, while offering limited or negative benefits in highly clustered configurations.**

I. Problem Description

In this project, flocking behavior is simulated using an extended boids model. The goal is to implement and accelerate this model on the GPU and to compare different implementation strategies while preserving identical or at least similar agent behavior.

The application supports multiple boid types (basic flock members, predators, static obstacles) and interactive control (mouse attraction/repulsion), and it can run both in 2D and 3D with either periodic or bouncing boundary conditions.

This section first introduces the original boids concept and then describes the specific model and equations used in this work.

A. Boids and flocking behavior

Boids (*bird-oid objects*) were introduced by Reynolds as an artificial life model for simulating the collective motion of birds, fish, and other animal groups [1].

The key idea is that each individual agent follows only simple local rules, yet the group as a whole exhibits complex emergent flocking behavior.

In the basic model, each boid reacts only to its neighbors within a limited perception radius and applies three canonical steering rules: separation, alignment, and cohesion [2].

- **Separation**: steer away from nearby flockmates to avoid collisions.
- **Alignment**: steer towards the average heading (velocity) of nearby flockmates.
- **Cohesion**: steer towards the average position (center of mass) of nearby flockmates.

Let $\mathbf{x}_i$ and $\mathbf{v}_i$ denote position and velocity of boid i and let $\mathcal{N}_i$ be the set of its neighbors (within a vision range). Typical formulations of the three steering rules use

$$\mathbf{f}_i^{\text{sep}} \propto -\sum_{j\in\mathcal{N}_i} \frac{\mathbf{x}_j - \mathbf{x}_i}{\|\mathbf{x}_j - \mathbf{x}_i\|}, \tag{1}$$

$$\mathbf{f}_i^{\text{ali}} \propto \left(\frac{1}{|\mathcal{N}_i|}\sum_{j\in\mathcal{N}_i} \mathbf{v}_j\right) - \mathbf{v}_i, \tag{2}$$

$$\mathbf{f}_i^{\text{coh}} \propto \left(\frac{1}{|\mathcal{N}_i|}\sum_{j\in\mathcal{N}_i} \mathbf{x}_j\right) - \mathbf{x}_i, \tag{3}$$

where the proportionality constants are application-specific weights or parametrized "gains". The full steering force is typically a weighted sum of these components.

More complex boids models augment these rules with additional behaviors such as obstacle avoidance, various boid types and their interactions, etc.

B. Boids model used in this work

The implementation used in this project is an extended boids model that supports multiple agent types, interactive user control, rigid spherical obstacles, and switchable boundary conditions.

The simulation maintains the following agent types:

- **Basic boids**: flock members that follow separation, alignment, cohesion, target attraction, obstacle avoidance, predator avoidance, and wall interaction rules.
- **Predator boids**: agents that chase basic boids, governed by a stamina-based state machine and subject to the same environmental interactions (obstacles, walls, etc.).
- **Obstacle boids**: static spherical obstacles that other boids must avoid and bounce off.

For the mathematical description of the model, several globally defined variables and notational conventions are used.

The notation

$$\widehat{\mathbf{v}} = \frac{\mathbf{v}}{\|\mathbf{v}\| + \varepsilon}$$

denotes a normalized vector, where a small constant ε is added to avoid division by zero.

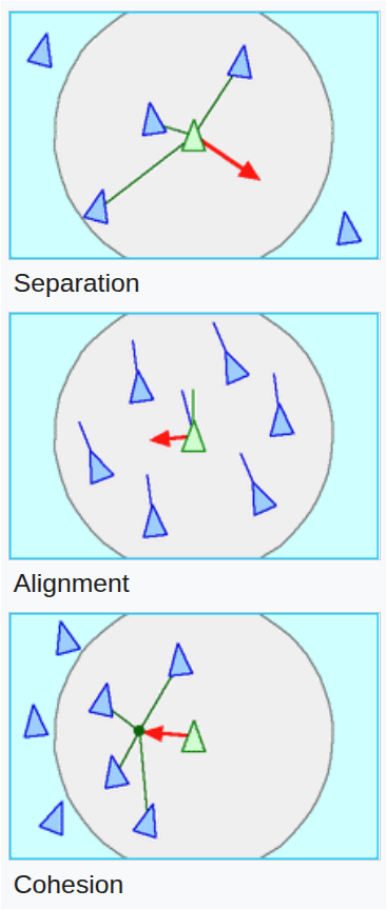


Figure 1: Conceptual illustration of the three basic boids rules: separation, alignment, and cohesion. [2]

The simulation domain is an axis-aligned box $[0, W_x] \times [0, W_y]$ in 2D or $[0, W_x] \times [0, W_y] \times [0, W_z]$ in 3D, where the upper bounds can be defined arbitrarily.

The maximum velocity $v_{\max}$ that any boid can reach is defined as

$$v_{\max} = 0.1\, W_{\max}, \quad \text{where} \quad W_{\max} = \max(W_x, W_y, W_z).$$

where the factor 0.1 was chosen empirically based on experimental tuning.

Based on this value, the base force magnitude is defined as

$$F_{\text{base}} = w_F\, v_{\max},$$

where the multiplier w_F can be tuned from the range $\langle 0, 1 \rangle$. This definition ensures that the overall behavior of the simulation remains consistent across different world sizes.

Furthermore, the maximum possible distance between any two points in the current simulation domain is denoted by $d_{\max}$.

The following paragraphs define the boid behavior used in this project in mathematical detail.

*1) **Basic Boids**:* As already mentioned, basic boids are the only agent type in the simulation that follows the flocking rules.

For each basic boid i, the simulation stores the following 3D vectors:

- position $\mathbf{x}_i^{\text{b}}$,
- velocity $\mathbf{v}_i^{\text{b}}$,
- acceleration $\mathbf{a}_i^{\text{b}}$,
- a local target point $\mathbf{t}_i^{\text{b}}$.

Moreover, all basic boids share a global radius r^{b} and a vision range $r_{\text{vis}}^{\text{b}}$, which is defined as

$$r_{\text{vis}}^{\text{b}} = d_{\max}\, w_{\text{vis}}^{\text{b}},$$

where $w_{\text{vis}}^{\text{b}}$ is a parameter specifying the fraction of the world that a boid can perceive at once.

At the start of the simulation, each basic boid is initialized with a random position within the domain, a small random velocity, zero acceleration, and a common target point located at the center of the current world.

Each simulation step for a basic boid i consists of three stages:

- (i) **Reset acceleration:** $\mathbf{a}_i^{\text{b}} = \mathbf{0}$.
- (ii) **Evaluate behavioral rules:** the rules described in paragraphs I-B1a to I-B1g.
- (iii) **Apply generic dynamics and collision handling:** described in the subsequent paragraphs, in the order in which they are applied.

For a basic boid i with position $\mathbf{x}_i^{\text{b}}$ and velocity $\mathbf{v}_i^{\text{b}}$, the implementation considers a neighborhood $\mathcal{N}_i^{\text{b}}$ consisting of other basic boids within its vision range. The exact neighbor selection strategy (naive all-pairs or grid-based) depends on the chosen implementation.

Let

$$d_{ij} = \|\mathbf{x}_j^{\text{b}} - \mathbf{x}_i^{\text{b}}\|$$

denote the distance between boids i and j, and let

$$r_{\text{p}}^{\text{b}} = 4r^{\text{b}}$$

be the protected range of a basic boid. The size of this range was selected empirically based on experimental tuning.

a) Separation: All neighbors in $\mathcal{N}_i^{\text{b}}$ contribute to the separation force

$$\mathbf{f}_i^{\text{s}} = -\sum_{j \in \mathcal{N}_i^{\text{b}}} \frac{\mathbf{x}_j^{\text{b}} - \mathbf{x}_i^{\text{b}}}{d_{ij}}, \tag{4}$$

which steers the boid away from nearby flockmates to prevent collisions.

b) Cohesion: Boids that are in $\mathcal{N}_i^{\text{b}}$ but outside the protected range r_{p}^{b} contribute to a cohesion force

$$\mathbf{f}_i^{\text{c}} = \frac{1}{|\mathcal{C}_i|} \sum_{j \in \mathcal{C}_i} \left(\mathbf{x}_j^{\text{b}} - \mathbf{x}_i^{\text{b}}\right), \qquad \mathcal{C}_i = \{j \in \mathcal{N}_i^{\text{b}} \mid d_{ij} > r_{\text{p}}^{\text{b}}\}. \tag{5}$$

This term pulls the boid toward the local center of mass of its neighbors.

c) Alignment: All neighbors in $\mathcal{N}_i^{\text{b}}$ contribute to the alignment force

$$\mathbf{v}_i^{\text{avg}} = \frac{1}{|\mathcal{N}_i^{\text{b}}|} \sum_{j \in \mathcal{N}_i^{\text{b}}} \mathbf{v}_j^{\text{b}}, \qquad \mathbf{f}_i^{\text{a}} = \mathbf{v}_i^{\text{avg}} - \mathbf{v}_i^{\text{b}}. \tag{6}$$

This force encourages the boid to match the average velocity of nearby flockmates.

d) Target attraction: Each basic boid is associated with a target point $\mathbf{t}_i^{\mathrm{b}}$. The displacement vector toward the target is computed as

$$\mathbf{d}_i^{\mathrm{t}} = \mathbf{t}_i^{\mathrm{b}} - \mathbf{x}_i^{\mathrm{b}}.$$

Based on the distance to the target, a scalar target force magnitude is defined as

$$f_i^{\mathrm{t}} = \frac{\|\mathbf{d}_i^{\mathrm{t}}\|^2}{10\, d_{\max}}, \tag{7}$$

where $d_{\max}$ denotes the maximum possible distance between two points in the current simulation domain. As a result, boids that are farther from their target are attracted more strongly. This force formulation was selected empirically based on experimental tuning.

e) Cruising speed: In addition, the implementation defines a desired cruising speed $v_{\mathrm{b}}^{\mathrm{cr}}$ and computes a force that steers the boid toward the corresponding cruising velocity:

$$\mathbf{v}_i^{\mathrm{cr}} = v_{\mathrm{b}}^{\mathrm{cr}}\, \widehat{\mathbf{v}}_i^{\mathrm{b}}, \qquad \mathbf{f}_i^{\mathrm{cr}} = \mathbf{v}_i^{\mathrm{cr}} - \mathbf{v}_i^{\mathrm{b}}. \tag{8}$$

This term encourages boids to maintain the prescribed cruising speed.

f) Weighted steering force: All steering contributions are first converted to unit directions and then combined into a single steering force using a global base force F_{base} and per-behavior weights:

$$\mathbf{f}_i^{\mathrm{b}} = F_{\mathrm{base}}\Big(w_{\mathrm{sep}}^{\mathrm{b}}\, \widehat{\mathbf{f}}_i^{\mathrm{s}} + w_{\mathrm{coh}}^{\mathrm{b}}\, \widehat{\mathbf{f}}_i^{\mathrm{c}} + w_{\mathrm{ali}}^{\mathrm{b}}\, \widehat{\mathbf{f}}_i^{\mathrm{a}} + w_{\mathrm{tar}}^{\mathrm{b}}\, \widehat{\mathbf{f}}_i^{\mathrm{t}} + w_{\mathrm{cru}}^{\mathrm{b}}\, \widehat{\mathbf{f}}_i^{\mathrm{cr}}\Big). \tag{9}$$

The weights $w_{\mathrm{sep}}^{\mathrm{b}}$, $w_{\mathrm{coh}}^{\mathrm{b}}$, $w_{\mathrm{ali}}^{\mathrm{b}}$, and $w_{\mathrm{tar}}^{\mathrm{b}}$ can be chosen arbitrarily from the interval $\langle 0, 1\rangle$. The cruising speed weight is fixed to $w_{\mathrm{cru}}^{\mathrm{b}} = 0.1$, based on experimental tuning. The resulting steering force is accumulated into the basic boid acceleration as

$$\mathbf{a}_i^{\mathrm{b}} \leftarrow \mathbf{a}_i^{\mathrm{b}} + \mathbf{f}_i^{\mathrm{b}},$$

before predator avoidance is applied.

g) Predator avoidance: Basic boids also actively avoid predator boids. For a basic boid i and a predator boid k with position $\mathbf{x}_k^{\mathrm{p}}$, the displacement vector from the boid to the predator is defined as

$$\mathbf{d}_{ik}^{\mathrm{bp}} = \mathbf{x}_k^{\mathrm{p}} - \mathbf{x}_i^{\mathrm{b}},$$

with corresponding distance

$$d_{ik}^{\mathrm{bp}} = \|\mathbf{d}_{ik}^{\mathrm{bp}}\|.$$

Basic boids use their vision range $r_{\mathrm{vis}}^{\mathrm{b}}$ to detect nearby predators. From all predators within this range, a repulsive escape direction is accumulated as

$$\mathbf{e}_i = -\sum_k \frac{5}{d_{ik}^{\mathrm{bp}}}\, \widehat{\mathbf{d}}_{ik}^{\mathrm{bp}}. \tag{10}$$

Each normalized direction is weighted inversely by distance and scaled by an empirically chosen constant factor of 5, causing closer predators to contribute more strongly to the escape direction.

If at least one predator is detected, the normalized escape direction $\widehat{\mathbf{e}}_i$ is converted into an escape force by scaling it by three times the base force:

$$\mathbf{f}_i^{\mathrm{esc}} = 3.0\, F_{\mathrm{base}}\, \widehat{\mathbf{e}}_i. \tag{11}$$

If the squared magnitude of this escape force exceeds that of the previously accumulated steering force, i.e., $\|\mathbf{f}_i^{\mathrm{esc}}\|^2 > \|\mathbf{f}_i^{\mathrm{b}}\|^2$, the existing steering is discarded and replaced by the escape force. Otherwise, the escape force is added to the current acceleration. This mechanism ensures that predator avoidance dominates in situations of imminent danger.

h) Interaction points: This paragraph introduces generic interaction handling that temporarily overrides the standard steering behavior. The same interaction rules are applied to predator boids as well, using their respective parameter values.

During the simulation, a single interaction point can be activated, affecting all boids in the scene. Two interaction types are supported: attraction and repulsion.

When an interaction occurs, the acceleration of a basic boid j is first reset:

$$\mathbf{a}_j^{\mathrm{b}} = \mathbf{0}.$$

The displacement vector from the boid to the interaction point $\mathbf{x}^{\mathrm{i}}$ is then computed as

$$\mathbf{d}_j^{\mathrm{bi}} = \mathbf{x}^{\mathrm{i}} - \mathbf{x}_j^{\mathrm{b}}, \tag{12}$$

with distance $d_j^{\mathrm{bi}} = \|\mathbf{d}_j^{\mathrm{bi}}\|$. Using a global interaction force multiplier w^{i} (default value 500, selected empirically), a distance-based weight is defined as

$$w_j^{\mathrm{bi}} = w^{\mathrm{i}} \left(\frac{d_j^{\mathrm{bi}}}{d_{\max}}\right)^2. \tag{13}$$

The corresponding interaction force is then given by

$$\mathbf{f}_j^{\mathrm{bi}} = F_{\mathrm{base}}\, w_j^{\mathrm{bi}}\, \widehat{\mathbf{d}}_j^{\mathrm{bi}}. \tag{14}$$

For an attraction interaction, the force is applied toward the interaction point, whereas for repulsion it is applied in the opposite direction:

$$\text{Attract:} \quad \mathbf{a}_j^{\mathrm{b}} \leftarrow +\mathbf{f}_j^{\mathrm{bi}}, \tag{15}$$

$$\text{Repel:} \quad \mathbf{a}_j^{\mathrm{b}} \leftarrow -\mathbf{f}_j^{\mathrm{bi}}. \tag{16}$$

Because the acceleration is reset at the beginning of interaction handling, such events temporarily override all previously accumulated steering forces.

i) Obstacle avoidance and wall repulsion: Obstacle and wall avoidance are implemented as soft steering behaviors that steer boids away from nearby obstacles and domain boundaries.

For a basic boid i with position $\mathbf{x}_i^{\mathrm{b}}$ and radius r^{b}, and for each obstacle boid with center $\mathbf{x}_\ell^{\mathrm{o}}$ and radius r^{o}, the following quantities are computed:

$$\mathbf{d}_{i\ell}^{\mathrm{bo}} = \mathbf{x}_\ell^{\mathrm{o}} - \mathbf{x}_i^{\mathrm{b}}, \tag{17}$$

$$d_{i\ell}^{\text{bo - center}} = \|\mathbf{d}_{i\ell}^{\mathrm{bo}}\|, \tag{18}$$

$$d_{i\ell}^{\text{bo - surface}} = d_{i\ell}^{\text{bo - center}} - (r^{\mathrm{b}} + r^{\mathrm{o}}). \tag{19}$$

If the surface distance $d_{i\ell}^{\text{bo - surface}}$ exceeds the vision range $r_{\text{vis}}^{\text{b}}$, the obstacle is ignored. Otherwise, an exponential proximity weight is computed as

$$w_{i\ell}^{\text{bo}} = \exp\left(-0.1\, d_{i\ell}^{\text{bo - surface}}\right), \tag{20}$$

where the decay factor 0.1 was selected empirically.

All nearby obstacles contribute to a weighted avoidance direction and total weight:

$$\mathbf{u}_i^{\text{bo}} = -\sum_{\ell} w_{i\ell}^{\text{bo}}\, \widehat{\mathbf{d}}_{i\ell}^{\text{bo}}, \qquad W_i^{\text{bo}} = \sum_{\ell} w_{i\ell}^{\text{bo}}. \tag{21}$$

If at least one obstacle is present, the average weight

$$\overline{w}_i^{\text{bo}} = \frac{W_i^{\text{bo}}}{n_i^{\text{o}}},$$

where n_i^{o} denotes the number of obstacles within the vision range of boid i, and the normalized avoidance direction $\widehat{\mathbf{u}}_i^{\text{bo}}$ are used to apply a single steering contribution:

$$\mathbf{a}_i^{\text{b}} \leftarrow \mathbf{a}_i^{\text{b}} + F_{\text{base}}\, \overline{w}_i^{\text{bo}}\, w^{\text{ow}}\, \widehat{\mathbf{u}}_i^{\text{bo}}, \tag{22}$$

where w^{ow} is a global obstacle and wall avoidance multiplier that can be chosen arbitrarily (default value 500, selected empirically).

The same approach is used to implement soft repulsion from the domain boundaries when bouncing is enabled. For each basic boid i and for each wall along axis a (two per axis in a 2D or 3D box), the distance d_{ia}^{bw} from the boid surface to the wall is computed. If d_{ia}^{bw} is smaller than the vision range, an exponential weight

$$w_{ia}^{\text{bw}} = \exp\left(-0.3\, d_{ia}^{\text{bw}}\right)$$

is applied, where the decay factor 0.3 was selected empirically. The corresponding acceleration component is then adjusted as

$$a_{ia}^{\text{b}} \leftarrow a_{ia}^{\text{b}} + \sigma\, F_{\text{base}}\, w_{ia}^{\text{bw}}\, w^{\text{ow}}, \tag{23}$$

where $\sigma \in \{-1, +1\}$ encodes the direction away from the wall along the given axis, and a_{ia}^{b} denotes the acceleration of boid i in dimension a.

j) Dynamics, drag, noise, and integration: After all steering forces have been accumulated (including interaction points and obstacle and wall avoidance), the simulation applies drag and noise and then integrates the motion. For a basic boid i with position $\mathbf{x}_i^{\text{b}}$, velocity $\mathbf{v}_i^{\text{b}}$, and acceleration $\mathbf{a}_i^{\text{b}}$, the following steps are performed:

- **Linear drag:** A velocity-proportional drag term is applied via an acceleration that would bring the boid to rest in n_{stop} steps at maximum drag:

$$\mathbf{a}_i^{\text{b}} \leftarrow \mathbf{a}_i^{\text{b}} - \frac{\mathbf{v}_i^{\text{b}}}{\Delta t\, n_{\text{stop}}}\, \gamma, \tag{24}$$

  where $\gamma \in \langle 0, 1\rangle$ is the drag coefficient and Δt is the simulation time step.
- **Steering noise:** The steering direction is randomized by blending the current acceleration direction with a random unit vector $\widehat{\mathbf{r}}$:

$$a_i^{\text{b}} = \|\mathbf{a}_i^{\text{b}}\|, \tag{25}$$

$$\mathbf{b}_i^{\text{b}} = (1-\eta)\, \widehat{\mathbf{a}}_i^{\text{b}} + \eta\, \widehat{\mathbf{r}}, \tag{26}$$

$$\mathbf{a}_i^{\text{b}} \leftarrow a_i^{\text{b}}\, \widehat{\mathbf{b}}_i^{\text{b}}, \tag{27}$$

  where $\eta \in \langle 0, 1\rangle$ is a noise parameter. Only the steering direction is perturbed, while the magnitude of the acceleration is preserved.
- **Velocity and position integration:** Velocity and position are integrated explicitly:

$$\mathbf{v}_i^{\text{b}} \leftarrow \mathbf{v}_i^{\text{b}} + \mathbf{a}_i^{\text{b}}\, \Delta t, \tag{28}$$

$$\mathbf{x}_i^{\text{b}} \leftarrow \mathbf{x}_i^{\text{b}} + \mathbf{v}_i^{\text{b}}\, \Delta t. \tag{29}$$

  After integration, the boid speed is clamped:

$$v_i^{\text{b}} = \|\mathbf{v}_i^{\text{b}}\|, \quad \mathbf{v}_i^{\text{b}} \leftarrow \begin{cases} \mathbf{v}_i^{\text{b}} \cdot \dfrac{v_{\max}^{\text{b}}}{v_i^{\text{b}}}, & v_i^{\text{b}} > v_{\max}^{\text{b}}, \\ \mathbf{v}_i^{\text{b}} \cdot \dfrac{v_{\min}^{\text{b}}}{v_i^{\text{b}} + \varepsilon}, & v_i^{\text{b}} < v_{\min}^{\text{b}}, \\ \mathbf{v}_i^{\text{b}}, & \text{otherwise}, \end{cases} \tag{30}$$

  where the minimum and maximum speeds are defined as $v_{\min}^{\text{b}} = v_{\max}\, w_{\text{min-vel}}^{\text{b}}$ and $v_{\max}^{\text{b}} = v_{\max}\, w_{\text{max-vel}}^{\text{b}}$. Here, $v_{\max}$ is the global maximum velocity for the current world size, and $w_{\text{min-vel}}^{\text{b}}$ and $w_{\text{max-vel}}^{\text{b}}$ are tunable weights from the interval $\langle 0, 1\rangle$.

If the simulation is configured in 2D, the z-components of both position and velocity are explicitly set to zero after integration.

k) Wall collisions: The position of a basic boid i, $\mathbf{x}_i^{\text{b}}$, is constrained to remain within the simulation domain, e.g. $[r^{\text{b}}, W_x - r^{\text{b}}]$ along the x-axis, and analogously for the remaining dimensions. If a boid crosses a wall, its position is projected back onto the domain boundary and its velocity is reflected with respect to the wall normal. This reflection is implemented by inverting the velocity component along the corresponding axis and scaling the entire velocity vector by a bounce factor $\beta \in \langle 0, 1\rangle$.

If bouncing is disabled, periodic boundary conditions are applied instead:

$$x_i^{\text{b}} \leftarrow \begin{cases} x_i^{\text{b}} + W_x, & x_i^{\text{b}} < 0, \\ x_i^{\text{b}} - W_x, & x_i^{\text{b}} \geq W_x, \\ x_i^{\text{b}}, & \text{otherwise}, \end{cases} \tag{31}$$

and analogously for the y-coordinate (and the z-coordinate in 3D).

l) Obstacle collisions: For collisions with obstacles, the simulation checks whether the distance between the boid center $\mathbf{x}_i^{\text{b}}$ and the obstacle center $\mathbf{x}_\ell^{\text{o}}$ is smaller than the sum of their radii. If

$$d_{i\ell}^{\text{bo - center}} < r^{\text{b}} + r^{\text{o}}, \tag{32}$$

the boid overlaps the obstacle. In this case, the boid is displaced along the normalized center-to-center direction $\widehat{\mathbf{d}}_{i\ell}^{\text{bo - center}}$ by the minimum amount required to resolve the overlap (plus a small ε), and its velocity is reflected about the obstacle surface normal:

$$\mathbf{x}_i^{\text{b}} \leftarrow \mathbf{x}_i^{\text{b}} + \widehat{\mathbf{d}}_{i\ell}^{\text{bo - center}} \left(-d_{i\ell}^{\text{bo - surface}} + \varepsilon\right), \tag{33}$$

$$\mathbf{v}_i^{\text{b - ref}} = \mathbf{v}_i^{\text{b}} - 2\left(\mathbf{v}_i^{\text{b}} \cdot \widehat{\mathbf{d}}_{i\ell}^{\text{bo - center}}\right) \widehat{\mathbf{d}}_{i\ell}^{\text{bo - center}}, \tag{34}$$

$$\mathbf{v}_i^{\text{b}} \leftarrow \beta\, \mathbf{v}_i^{\text{b - ref}}. \tag{35}$$

*2) **Predator Boids**:* Predator boids combine cruising behavior with target chasing and a stamina-based mechanism.

For each predator boid k, the simulation stores the following 3D vectors:

- position $\mathbf{x}_k^{\mathrm{p}}$,
- velocity $\mathbf{v}_k^{\mathrm{p}}$,
- acceleration $\mathbf{a}_k^{\mathrm{p}}$.

In addition, each predator boid maintains its current stamina level s_k^{p}, a resting-state flag r_k^{p}, and information about its currently selected target boid.

A global maximum stamina $s_{\max}^{\mathrm{p}}$ is defined, together with stamina drain and recovery rates $s_{\mathrm{drain}}^{\mathrm{p}}$ and $s_{\mathrm{rec}}^{\mathrm{p}}$, respectively.

Predator boids also have a global radius r^{p} and a vision range $r_{\mathrm{vis}}^{\mathrm{p}}$, defined as

$$r_{\mathrm{vis}}^{\mathrm{p}} = d_{\max}\, w_{\mathrm{vis}}^{\mathrm{p}},$$

where $w_{\mathrm{vis}}^{\mathrm{p}}$ specifies the fraction of the world that a predator can perceive at once.

A predator boid is created when required, with a random or user-specified initial position, a small random velocity, and zero acceleration.

Each simulation step for a predator boid k consists of three stages:

- (i) **Reset acceleration:** $\mathbf{a}_k^{\mathrm{p}} = \mathbf{0}$.
- (ii) **Evaluate behavioral rules:** the rules described in paragraphs I-B2a and I-B2b.
- (iii) **Apply generic dynamics and collision handling:** analogous to the procedures used for basic boids, but with predator-specific parameters.

a) Cruising: First, a cruising force analogous to that of basic boids is applied:

$$\mathbf{v}_k^{\mathrm{cr}} = v_{\mathrm{p}}^{\mathrm{cr}}\, \widehat{\mathbf{v}}_k^{\mathrm{p}}, \tag{36}$$

$$\mathbf{f}_k^{\mathrm{cr}} = F_{\mathrm{base}} \left(\mathbf{v}_k^{\mathrm{cr}} - \mathbf{v}_k^{\mathrm{p}}\right) w_{\mathrm{p}}^{\mathrm{cr}}, \tag{37}$$

$$\mathbf{a}_k^{\mathrm{p}} \leftarrow \mathbf{a}_k^{\mathrm{p}} + \mathbf{f}_k^{\mathrm{cr}}, \tag{38}$$

where the cruising weight is fixed to $w_{\mathrm{p}}^{\mathrm{cr}} = 0.5$, and $v_{\mathrm{p}}^{\mathrm{cr}}$ specifies the desired cruising speed.

b) Chasing and stamina: At each simulation step, a predator boid that is not resting selects its target as the closest basic boid within its vision range.

If predator k has a valid target basic boid i, is not resting, and has positive stamina ($s_k > 0$), it accelerates toward the target:

$$\mathbf{d}_k^{\mathrm{tp}} = \mathbf{x}_i^{\mathrm{b}} - \mathbf{x}_k^{\mathrm{p}}, \tag{39}$$

$$d_k^{\mathrm{tp}} = \|\mathbf{d}_k^{\mathrm{tp}}\|, \tag{40}$$

$$\mathbf{a}_k^{\mathrm{p}} = \left(d_k^{\mathrm{tp}}\right)^2 \widehat{\mathbf{d}}_k^{\mathrm{tp}}, \tag{41}$$

so that the acceleration magnitude scales with the squared distance to the target.

During chasing, stamina is reduced according to

$$s_k \leftarrow s_k - s_{\mathrm{drain}}^{\mathrm{p}}\, \Delta t, \tag{42}$$

where Δt denotes the simulation time step.

When stamina reaches zero, the predator enters a resting state ($r_k^{\mathrm{p}} = 1$), and its stamina is clamped to zero. While resting, the predator no longer chases and instead recovers stamina:

$$s_k \leftarrow s_k + s_{\mathrm{rec}}^{\mathrm{p}}\, \Delta t. \tag{43}$$

Once s_k exceeds the maximum stamina $s_{\max}^{\mathrm{p}}$, the predator returns to the active state ($r_k^{\mathrm{p}} = 0$), and s_k is clamped to $s_{\max}^{\mathrm{p}}$.

At the end of each predator update, the stored target index and target distance are reset to sentinel values in preparation for the next simulation step.

*3) **Obstacle Boids**:* Obstacle boids represent static objects that other boid types attempt to avoid and bounce off when avoidance fails. Consequently, obstacle boids do not exhibit any dynamic behavior or steering rules of their own.

For each obstacle boid m, the simulation stores the following 3D vector:

- position $\mathbf{x}_m^{\mathrm{o}}$.

All obstacle boids share a common global radius r^{o}.

II. Implementation

This section focuses solely on the project implementation. For notes on running the app and the exact source codes, turn to the code base and included readme.md file.

The behavioral model described in the previous section is shared by four different simulation implementations. They differ only in how neighborhoods are computed and where the computation runs; all parameters (vision ranges, weights, radii, etc.) and thus the effective neighbor accuracy are kept consistent.

- **Sequential naive:** a CPU implementation that iterates over all boid pairs to determine neighborhoods and evaluate forces. This has $O(N^2)$ complexity.
- **Sequential grid-based:** a CPU implementation that partitions the domain into a uniform grid and searches only nearby cells for neighbors, reducing the cost of neighborhood queries while producing the same neighborhoods (up to the configured neighbor limit).
- **Parallel naive (GPU):** a CUDA implementation where each boid is processed in a separate thread, but neighbors are still found by iterating over all boids. This preserves the naive neighbor accuracy while exploiting data parallelism.
- **Parallel grid-based (GPU):** a CUDA implementation that uses a uniform grid for neighborhood queries on the GPU. This reduces the asymptotic complexity of neighbor search while preserving the same behavioral rules and parameters as the other variants.

The implementation of these four versions only contains the simulation step itself (update). They are wrapped by a common framework that runs on CPU and that can run on two base modes: interactive mode that includes GUI and experiment lab mode for designing modular experiments and conduct measurements.

First, it is reasonable to describe the framework and later describe the versions themselves.

A. Application Architecture

The project is run with two arguments: mode in which the program will run and path to configuration directory.

First, based on the selected mode: interactive or experiment lab, the respective manager object is created (main.cpp): Application object (App.hpp) for the interaction mode and ExperimentLab object (ExperimentLab.hpp). As the objects are created they are also initialized and this initialization for both starts with configuration loading based on the configuration directory path provided as initial argument.

The following subsection describe each part of this.

*1) **Configuration**:* This part describes the general approach to configuration used throughout the project. Although several distinct configuration types are defined (covering experiments, algorithm variants, and the initial simulation state), they all share a common internal structure and are processed by the same configuration pipeline. This unified design ensures consistent handling, validation, and presentation of configuration data across all components of the system.

Each configuration file follows a strict and uniform structure. It starts with a top-level identifier that uniquely names the configuration and then defines all parameters grouped into clearly labeled categories. These categories are used only for organization and readability.

Inside each category, parameters are listed individually and described in a declarative, human-readable form. Each parameter specifies its type, internal name, user-facing label, description, default value, and optional constraints on the allowed values. This keeps configuration files easy to understand and edit, while still providing all information needed for automatic loading and validation.

A simplified example of the structure is shown below:

```
{
  "id": "parallel",
  "parameters": {
    "Behavior weights (basic)": [
      {
        "type": "number",
        "name": "alignmentBasic",
        "default": 20.0,
        "min": 0.0,
        "max": 100.0
      }, ...
    ], ...
  }
}
```

The important part of the configuration handling is the `Config` class (`Config.hpp`), which acts as the central container for all configuration parameters used in the system. It stores parameters in a typed, name-addressable form, provides constant-time lookup via an internal index, and exposes type-safe accessors for numeric, boolean, and string values. In addition, it maintains grouping metadata that associates parameters with named categories while preserving insertion order, enabling structured presentation and processing. The class also supports global reset of parameters to their default values, making it suitable as a reusable base for concrete configuration definitions.

The building stone of the configuration class itself is yet another class: `ConfigParameter` (`ConfigParameter.hpp`), which represents a single strongly-typed configuration entry together with its associated metadata. Each parameter encapsulates its name, user-facing label, textual description, type information, current value, and default value. In addition, it defines the allowed value domain through explicit range constraints and provides rendering hints intended for user interface generation. The class offers factory methods for constructing numeric, binary, string, and enumeration parameters, along with type-safe accessors that enforce correctness at runtime. This design makes `ConfigParameter` a self-contained, expressive building block that cleanly separates parameter definition, validation, and presentation concerns.

There is an option for loading one or multiple configuration files at once. When a configuration directory is provided, the directory is checked and all json files are then loaded into an array of Config objects (MultipleConfigLoader.hpp). Each file is loaded into memory (JsonLoader.hpp) and parsed based on the required form (ConfigJsonParser.hpp) by enforcing the top level structure and loading and parsing an arbitrary number of parameters (ParameterJsonParser.hpp) across various groups.

*2) **Interactive Mode**:* When the interactive mode is selected, the application object is created first in `main.cpp`. During construction, the application is initialized and, if successful, subsequently executed via the main run loop.

The initialization phase is responsible for preparing all core subsystems required for interactive execution. During this phase, the `VersionManager` is constructed and populated by loading all available version configurations from disk, and the initial simulation state is created as a `SimState` instance. The windowing platform is then initialized through the `GUI` object, followed by the creation of the graphical user interface context. After successful GUI initialization, the active simulation configuration is retrieved as a `Config` instance based on the initially selected version. Once all these components are successfully set up, the application is marked as initialized and ready to enter the main execution loop.

- The version manager (`VersionManager.hpp`) stores and manages multiple version-specific simulation configurations, provides a list of available versions, and exposes a selectable version parameter that allows the active configuration to be switched at runtime.
- The simulation state (`SimState.hpp`) aggregates all runtime data of the simulation, binding configuration parameters to mutable runtime fields, storing the current world and population state, and providing reset utilities for restoring the simulation to its initial or newly loaded configuration. It also owns the current boid population, which is stored in a dedicated `Boids` structure.
  The `Boids` structure (`Boids.hpp`) represents the full simulation population using a structure-of-arrays (SoA) layout, with separate buffers for basic boids, predator boids, and obstacle boids. It stores per-boid attributes such as position, velocity, acceleration, and interaction-

specific data, and provides helper functions for resizing and clearing populations during simulation and experiment setup.
- The graphical user interface (`GUI.hpp`) is responsible for initializing the windowing and rendering platform, managing the per-frame GUI lifecycle, rendering the simulation world and control panels, and translating user input into interaction data used by the simulation.

The running loop represents the main execution cycle of the application. While the GUI subsystem indicates that the application is active (is not closed), a new frame is started, user input is processed, and the simulation state is updated accordingly. During each iteration, the application checks for changes in the selected simulation version and reloads the corresponding configuration if needed. The simulation update step is then executed, followed by handling user-triggered reset actions for either simulation state or configuration values. Finally, the current simulation state and configuration are rendered through the GUI, the frame is finalized, and the loop continues until the application is closed, after which all GUI resources are cleanly shut down.

The simulation update (`SimulationUpdate.hpp`) defines the per-frame update routine, applying user interactions, regulating the boid population, advancing the simulation when not paused, and updating the global simulation time counter.
- The interaction system (`InteractionSystem.hpp`) processes mouse input from the GUI and translates it into runtime effects on the simulation state, such as spawning or removing entities, modifying interaction forces, and updating the active interaction mode.
- The boid population regulation (`BoidPopulationRegulation.hpp`) ensures that the actual number of simulated entities matches the target population values by spawning or removing boids as needed, and handles global population-level control actions such as clearing obstacles.
- The simulation step dispatcher (`SimulationStep.hpp`) selects and executes the appropriate simulation implementation based on the currently active version, constructing the corresponding parameter object and invoking the matching update routine.

The next level of implementation (each version simulation step) will be described later in this section.

*3) **Experiment Laboratory Mode**:* When the experiment laboratory mode is selected, an `ExperimentLab` object is created (`main.cpp`). Its purpose is to execute predefined experiment scenarios in a non-interactive, batch-oriented manner, without initializing any graphical interface.

During construction (`ExperimentLab.hpp`), the experiment laboratory performs its initialization by loading all available version configurations through the `VersionManager` and creating a default `SimState` from the initial simulation state configuration. Based on the version specified in this default state, the corresponding default simulation configuration (`Config`) is also loaded. Once these components are prepared, the laboratory is marked as initialized and ready to run experiments.

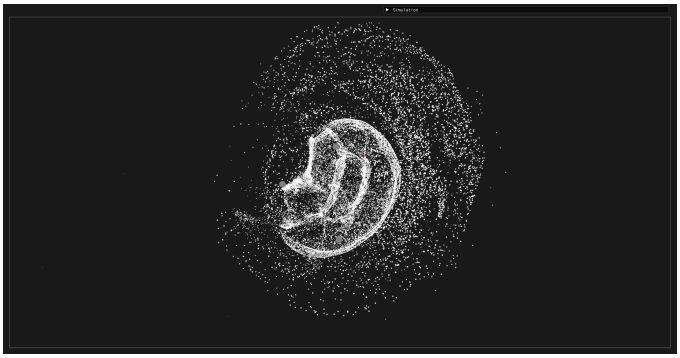


Figure 2: Showcase of the visual output of the interactive simulation GUI.

After initialization, the laboratory loads all experiment scenario configurations from the specified directory and executes them sequentially. For each experiment, the default simulation state and configuration are copied and selectively overridden according to experiment-specific parameters. The experiment pipeline then iterates over requested boid population sizes and simulation versions, initializes the simulation state for each scenario, and repeatedly executes the simulation step while collecting performance or state data via experiment-specific hooks. This mode enables systematic, reproducible evaluation of simulation behavior and performance across multiple configurations and implementations. (`ExperimentLab.hpp`)

B. Simulation Step Versions

In this part, each of the aforementioned versions of the simulation step will be described. As already mentioned, for each version, its parameters were first destilated, calculated, and processed from the SimState and SimConfig. That will also be described for each version.

*1) **Naive Sequential Simulation Step**:* This is the first and simplest CPU implementation of the simulation step. Its naive nature comes from explicitly iterating over all pairs of boids to determine neighborhoods and evaluate interaction forces, which results in a time complexity of $\mathcal{O}(N^2)$.

Before the simulation step itself is executed, all required parameters are prepared in a dedicated parameter object (`SequentialNaiveParameters.hpp`). This initialization phase gathers values from both the current simulation state and the active simulation configuration, converts percentage-based parameters into normalized weights, and precomputes commonly used quantities such as world dimensions, distance limits, vision ranges, speed limits, and force magnitudes. By consolidating all derived constants and references to boid data into a single structure, the subsequent simulation step can operate efficiently without repeatedly accessing configuration objects or performing redundant calculations.

The implementation of the simulation step itself (`SequentialNaive.hpp`) closely follows the algorithmic description presented in Section I.

An important implementation detail is the way distance vectors, and consequently distances, are computed to correctly support both supported boundary modes: wall bouncing and

periodic wrapping (`sim.hpp`). For this purpose, all spatial differences are evaluated using a periodic delta formulation. In wrapping mode, the distance between two coordinates d along a given axis of size L is adjusted as

$$\Delta d = \begin{cases} d - L, & d > \frac{1}{2}L, \\ d + L, & d < -\frac{1}{2}L, \\ d, & \text{otherwise.} \end{cases}$$

This ensures that the shortest displacement across periodic boundaries is always used. In bouncing mode, no such correction is applied and the raw coordinate difference is used directly. The resulting three-dimensional distance vector is then used consistently for neighborhood checks and force evaluation, guaranteeing correct behavior under both boundary conditions. The very same approach is used in all the other version implementations as well.

*2) **Grid-Based Sequential Simulation Step**:* The grid-based sequential simulation step is an optimized CPU implementation that improves upon the naive approach by reducing the cost of neighborhood queries. Instead of explicitly iterating over all pairs of boids, the simulation domain is partitioned into a uniform spatial grid, which restricts neighborhood searches to nearby cells. This replaces the global all-to-all search with local queries whose cost depends on the number of boids per cell rather than on the total population size.

From a complexity perspective, building the spatial grid requires a single linear pass over all boids, and each boid then inspects only a limited set of neighboring cells. Under uniform spatial distributions, this results in near-linear $\mathcal{O}(N)$ behavior in practice, as opposed to the $\mathcal{O}(N^2)$ complexity of the naive approach. However, in this particular use case boids tend to form dense groups, which increases local cell occupancy and reduces the effectiveness of spatial partitioning. As a result, while the grid-based method still provides a measurable speedup, the theoretical improvement is not fully realized in typical simulation scenarios.

As in the naive version, all parameters required for the simulation step are first collected and precomputed in a dedicated parameter object (`SequentialParameters.hpp`). In addition to the common physical, behavioral, and world-related parameters, this initialization phase also derives spatial grid parameters such as the cell size and the number of cells along each world dimension. The cell size is chosen to fully cover the maximum interaction radius, ensuring that all relevant neighbors are contained either in the same cell or in immediately adjacent cells.

The simulation step itself (`Sequential.hpp`) follows the same algorithmic structure as the naive implementation. The key difference lies in neighborhood evaluation: a spatial grid (`SpatialGrid.hpp`) is rebuilt at the beginning of each simulation step, and boids are inserted into grid cells based on their positions. For each boid, potential neighbors are then gathered by querying only the surrounding cells (9 in 2D, 27 in 3D), significantly reducing the number of distance checks required.

Distance vector computation and boundary handling are performed in the same manner as in the naive implementation, including support for both wall bouncing and periodic wrapping. As a result, the grid-based version preserves identical simulation behavior while achieving substantially better performance for larger boid populations.

*3) **Naive Parallel Simulation Step**:* The naive parallel simulation step is a direct GPU counterpart to the naive sequential implementation. Conceptually, it follows the same algorithmic structure and performs the same all-to-all neighborhood evaluation, but distributes the computation across many GPU threads to exploit data-level parallelism. As a result, the underlying computational complexity remains $\mathcal{O}(N^2)$, but the constant factor is significantly reduced by parallel execution.

Before the simulation step is executed, all required parameters are prepared in a dedicated parameter object (`ParallelNaiveParameters.cuh`). This initialization phase mirrors the CPU version by deriving all simulation constants from the current simulation state and configuration, but additionally handles GPU-specific responsibilities. These include allocating device memory for all boid attribute buffers, transferring the current boid data from host to device, and configuring execution parameters such as the CUDA block size. Upon destruction of the parameter object, the updated boid data is copied back to the CPU and all device memory is released.

The simulation step itself (`ParallelNaive.cuh`) launches separate CUDA kernels for each boid type (except the obstacle type). One kernel processes all basic boids, and a second kernel processes all predator boids. In each case, the number of CUDA blocks is computed from the current boid count and the configured block size, such that each GPU thread is responsible for updating a single boid. Inside the kernels, each thread evaluates interactions against all other relevant boids in the system, replicating the naive all-pairs logic of the sequential version. All boid attributes and simulation parameters are stored in global device memory, from which each thread reads the required data and to which it writes the updated state.

As in the CPU implementations, distance computation and boundary handling are performed using the same periodic delta formulation to support both bouncing and wrapping boundary modes (`SimStepParallelUtils.cuh`). This ensures that the parallel naive version produces behavior identical to the sequential naive implementation, while leveraging GPU parallelism to achieve substantially higher throughput for larger populations.

*4) **Grid-Based Parallel Simulation Step**:* The grid-based parallel simulation step is the most optimized implementation in this project, combining GPU parallelism with spatial partitioning to significantly reduce the cost of neighborhood queries. Conceptually, it corresponds to the grid-based sequential version, but all major stages of grid construction and simulation execution are implemented as CUDA kernels operating on device memory.

Before each simulation step, all required parameters are prepared in a dedicated parameter object (`ParallelParameters.cuh`). In addition to deriving physical and behavioral constants and transferring boid data to the GPU, this phase also constructs all data structures needed for spatial partitioning on the device. A uniform grid is

defined over the simulation domain, with the cell size chosen to cover the maximum interaction radius. This guarantees that all relevant neighbors of a boid are located either in its own cell or in immediately adjacent cells, allowing neighborhood queries to remain local.

The simulation step orchestration (`Parallel.cuh`) consists of several distinct GPU stages, each implemented as one or more CUDA kernels and executed in a fixed sequence.

a) Grid reset: At the beginning of each step, the grid cell metadata is reset using the `kernelResetCells` kernel (`CellResetKernels.cuh`). This kernel initializes per-cell range markers and stores the 3D cell coordinates corresponding to each linear cell index. This step prepares the grid for rebuilding without reallocating memory.

b) Spatial hashing: For each boid type (basic, predator, and obstacle), the `kernelComputeHashes` kernel is launched (`HashesKernels.cuh`). Each thread processes one boid and computes the grid cell index from the boid's world position. This includes correct handling of both bouncing and periodic boundary modes. The resulting cell index is flattened into a linear hash, which is stored together with the original boid index.

c) Hash-based sorting: The boids are then sorted by their cell hashes using a GPU bitonic sort (`BoidSorting.cuh`, `SortKernels.cuh`). The sorting pipeline first pads the hash buffers to the next power of two and then performs a full bitonic sorting network. Sorting boids by cell hash ensures that boids belonging to the same grid cell occupy contiguous regions in memory, which is essential for efficient neighbor lookup.

d) Cell range construction: After sorting, the `kernelBuildCellRanges` kernel is used to compute the start and end indices of each grid cell within the sorted hash arrays (`CellResetKernels.cuh`). These index ranges define compact per-cell intervals that can later be traversed efficiently during neighborhood queries.

e) Simulation kernels: Once the grid structure is built, the actual simulation step is executed. Two simulation kernels are launched: one for basic boids and one for predator boids (`ParallelSimStepKernels.cuh`). As in the naive parallel version, each thread is responsible for updating a single boid. However, instead of iterating over all other boids, each thread queries only the grid cells surrounding its own cell (9 in 2D, 27 in 3D) and evaluates interactions against boids found within those cells. Obstacle boids participate in grid construction and neighbor queries but do not have an associated simulation kernel, as they are static.

All boid attributes, grid structures, and intermediate buffers are stored in global device memory throughout the simulation step. Distance computation and boundary handling use the same periodic delta formulation as in the CPU and naive GPU implementations, ensuring identical simulation behavior across all versions.

By combining spatial partitioning with massively parallel execution, this approach achieves substantially better scalability than the naive parallel version, especially for larger populations, while preserving correctness and consistency with the reference algorithm.

III. Experiment Methods

In accordance with the assignment requirements, a series of experiments was designed to evaluate both the performance and correctness of the implemented simulation variants.

Two types of experiments were defined. The first experiment measures the execution time of a single simulation step in order to compare the performance of individual implementations. The second experiment focuses on saving the simulation state, which serves as a means of verifying the correctness of the measured results.

In addition, three simulation scenarios were proposed. These scenarios differ in dimensionality, interaction complexity, and the presence of obstacles and predators. Finally, all experiments were executed on two different machines in order to compare performance across hardware platforms.

A. Experiment Types

All experiment types share a common execution pipeline. Each experiment is defined by a configuration file specifying the simulation scenario and experiment parameters. Based on this configuration, the experiment is executed over a specified range of basic boid counts.

For each subexperiment (with a specific number of basic boids), the simulation state is first initialized according to the selected scenario. This includes spawning the required number of basic boids, predators, and obstacles at predefined or random positions. The simulation is then advanced from this initial state for a fixed number of steps in order to reach a more stable configuration, for example allowing boids to form groups. After this preparation phase, the experiment is executed separately for each simulation version.

Before collecting measurements for a given version, a number of warm-up steps is performed. These steps are excluded from evaluation and serve to eliminate transient effects such as cache warming or initial state instability.

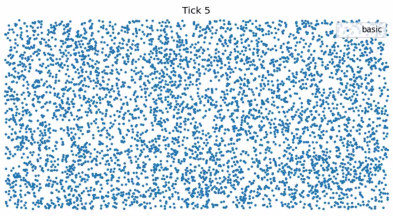
The first experiment type measures the execution time of the simulation step. Its purpose is to evaluate and compare the performance of different simulation versions and hardware platforms. For each version, scenario, and specific number of spawned basic boids, the measured execution times are averaged over multiple steps.

The second experiment type records the simulation state at each step. This experiment is intended to verify correctness by allowing visual inspection of the simulation behavior. For each version, scenario, and boid count, the simulation state is saved for every step and later converted into image sequences or videos showing the evolution of the system over time.

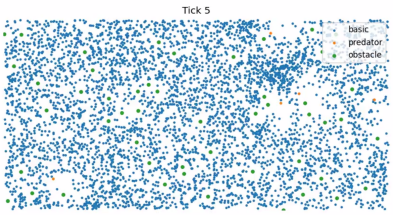
B. Simulation Scenarios

Three distinct simulation scenarios were selected to highlight different aspects of the simulation behavior and performance characteristics.

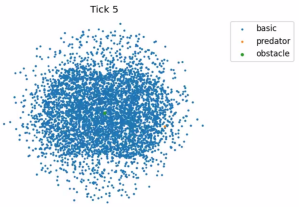
The first scenario is intentionally simple and is referred to as *Plain*. The simulation runs in two dimensions, and all interactions between basic boids are disabled, including separation, alignment, cohesion, and target attraction. Boids still



(a) Plain



(b) Plain with Obstacles



(c) 3D Complex

Figure 3: Visualization of the simulation state for the three evaluated scenarios. Visualized based on data collected with the state recording experiment.

interact with the world boundaries by bouncing off the walls and are spawned at random positions. As a result, the boids remain approximately uniformly distributed throughout the simulation domain, a situation in which grid-based approaches are expected to perform particularly well.

The second scenario, referred to as *Plain With Obstacles and Predators*, extends the first one by introducing additional entity types. In this case, 50 obstacles and 10 predators are randomly spawned. Basic boids interact with obstacles and predators according to the defined rules but still do not interact with other basic boids. This scenario is used to evaluate the impact of cross-type interactions on performance.

The third scenario, referred to as *3D Complex*, is significantly more complex. Basic boids are randomly spawned in a three-dimensional space with wrapping boundary conditions. One obstacle is placed at the center of the domain, and one predator is spawned at a random location. In this setup, basic boids interact both with each other and with other entity types. Due to the higher interaction complexity and the tendency of boids to form dense groups, the advantages of grid-based approaches are expected to diminish, and the overhead of spatial partitioning may even outweigh its benefits.

Also, for each simulation scenario, two distinct ranges for the number of spawned basic boids were considered. The first range, $\langle 100, 5000 \rangle$ with a step size of 100, was used to evaluate all implementation variants. The second range, $\langle 5000, 100000 \rangle$ with a step size of 5000, was used only for the GPU-based implementations, as testing the sequential versions in this range is not computationally feasible.

One simulation frame for each scenario and 5 000 basic boids is visualized in Figure 3.

C. Machines

The experiments were executed on two different machines in order to evaluate the behavior of the implementations across distinct hardware architectures, as required by the assignment.

The first machine, referred to as *Local*, is a personal laptop (ASUS TUF GAMING A15) used for development and local testing. Its relevant hardware specifications are as follows:

- CPU: AMD Ryzen 7 7435HS, 8 cores / 16 threads, x86_64 architecture, base frequency 400 MHz, boost frequency up to 4.55 GHz.
- RAM: 16 GB system memory.
- GPU: NVIDIA GeForce RTX 4060 with 8 GB of VRAM, CUDA version 13.0.

The second machine, referred to as *Server*, is a compute server provided by CTU and represents a more typical high-performance environment:

- CPU: Intel Xeon Silver 4114, 16 cores, x86_64 architecture, base frequency 2.20 GHz.
- RAM: 32 GB system memory.
- GPU: NVIDIA A16 with 16 GB of VRAM, CUDA version 12.9.

IV. Experiment Results

In this section, the results of the aforementioned experiments are presented and discussed from several perspectives.

All GPU-based experiments were executed using 1 024 threads per block.

A complete set of graphs is provided in the appendix.

A. CPU vs GPU Performance

This subsection focuses on a general comparison between the CPU (sequential) and GPU (parallel) implementations.

As shown in Figure 4, which reports the simulation step time for all implementations in the simple *Plain* scenario, the CPU-based versions fall behind the GPU implementations very quickly, even for relatively small numbers of basic boids. At 5 000 boids, the parallel implementations are nearly indistinguishable, both achieving runtimes of approximately 1 ms per simulation step. In contrast, the grid-based sequential implementation is roughly one order of magnitude slower, while the naive sequential implementation is slower by nearly two orders of magnitude.

Overall, the GPU implementations clearly outperform the CPU implementations. This trend is consistent across all tested scenarios and machines, as documented in the appendix.

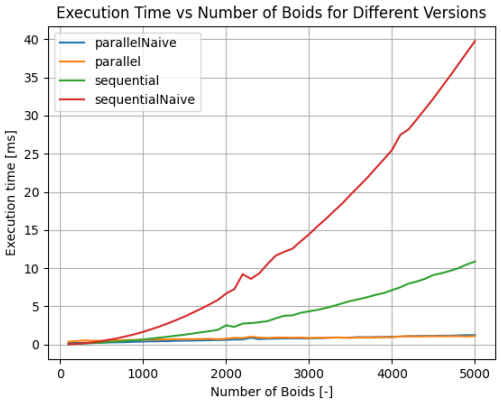


Figure 4: Scenario *Plain*, Machine *Local*, Range *(100–5 000, step 100)*

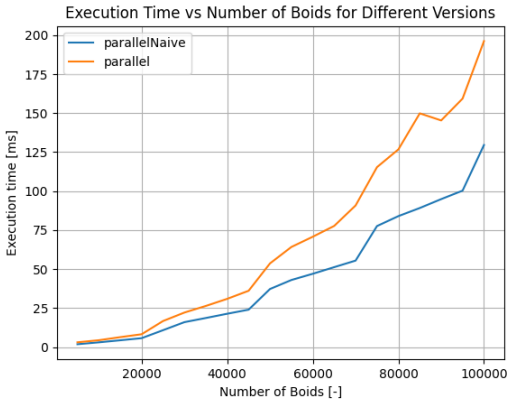


Figure 6: Scenario *3D Complex*, Machine *Local*, Range *(5 000–100 000, step 5 000)*

B. Naive vs Grid-Based Performance

This subsection compares the naive and grid-based implementations.

Figure 5 shows that, as expected, the naive parallel implementation performs worse than the grid-based parallel implementation in the *Plain* scenario. In this configuration, basic boids are evenly distributed across the simulation domain, allowing the spatial grid to effectively reduce the number of neighbor interactions.

In contrast, Figure 6, corresponding to the more complex *3D Complex* scenario, reveals two notable effects. First, both parallel and naive implementations exhibit higher runtimes than in the *Plain* scenario, because a larger number of boids fall within each other's vision range. Second, and more importantly, the grid-based implementation becomes slower than the naive one. This occurs because the benefits of spatial partitioning diminish when boids cluster into a small number of grid cells, while the overhead associated with grid construction and traversal remains.

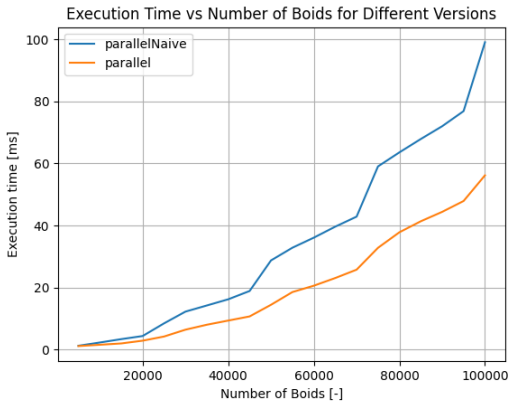


Figure 5: Scenario *Plain*, Machine *Local*, Range *(5 000–100 000, step 5 000)*

These observations are consistent with the expected behavior discussed earlier. For sequential implementations, the naive approach remains slower than the grid-based one; however, the performance gap is noticeably smaller in scenarios where boids are highly clustered. It is likely that with even higher boid counts or more extreme clustering parameters, the naive approach would eventually outperform the grid-based implementation in these cases as well.

C. Local vs Server

This subsection compares experiments executed on the local machine and on the server.

By comparing Figure 4, which shows results from the local machine, with Figure 7, which presents results obtained on the server, it can be seen that the previously identified trends are consistent across both hardware platforms for this configuration. This consistency also holds for all other tested scenarios and parameter settings.

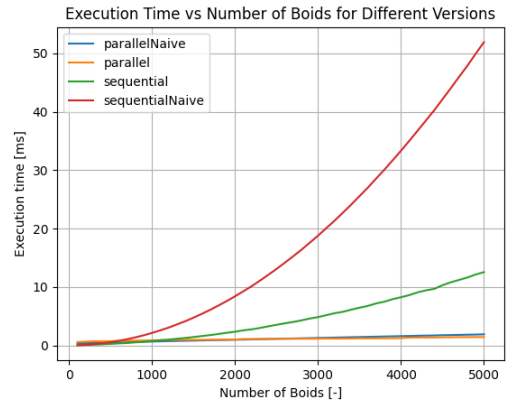


Figure 7: Scenario *Plain*, Machine *Server*, Range *(100–5 000, step 100)*

D. State Capturing Experiment

For each experimental configuration, a state capturing experiment was performed. This experiment produced data that were subsequently visualized, as illustrated in Figure 3, for every frame of every experiment run. The resulting visualizations

were then inspected to verify correctness and consistency of the observed behavior.

V. Conclusion

This project successfully implemented and evaluated four variants of an extended boids simulation, ranging from a naive sequential CPU version to an optimized grid-based GPU implementation. The results confirm that GPU parallelization is essential for achieving interactive or scalable performance with large boid populations. Grid-based neighborhood evaluation further improves performance in scenarios with relatively uniform spatial distributions, but its advantages diminish when boids form dense clusters, where grid construction overhead can outweigh its benefits.

Across all tested scenarios and hardware platforms, the simulation variants exhibited consistent behavior, validating the correctness of the parallel implementations with respect to the reference sequential model. While correctness is currently assessed visually by comparing simulation states, a more rigorous test could be performed by disabling randomness and automatically checking state consistency across versions over time. The experiment laboratory mode proved effective for systematic performance evaluation and reproducible benchmarking.

Possible future improvements include adaptive or hierarchical spatial partitioning to better handle clustered distributions, more efficient GPU sorting and grid construction strategies, and overlap of computation and data transfers to further reduce overhead. Extensions of the behavioral model, such as anisotropic perception or more complex agent interactions, could also be explored while leveraging the established parallel framework.

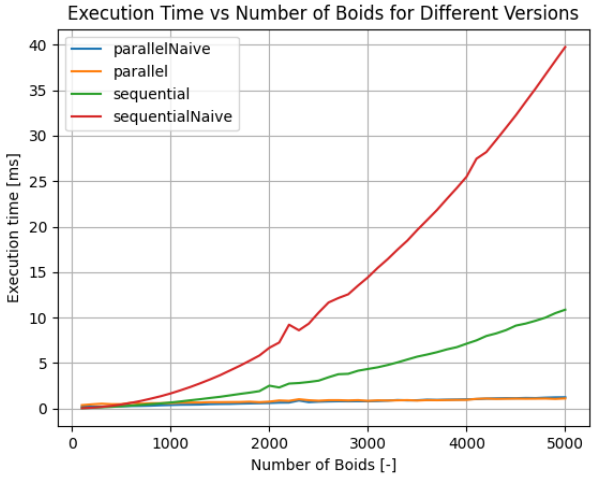
References

[1] C. W. Reynolds, “Flocks, herds, and schools: A distributed behavioral model,” in *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH ’87)*. ACM, 1987, pp. 25–34.

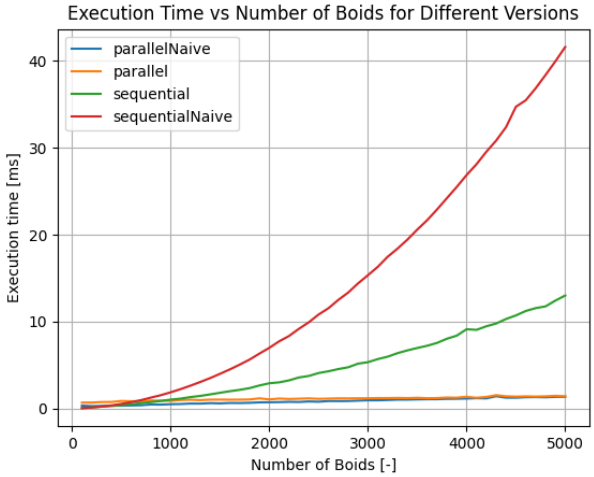
[2] Wikipedia contributors, “Boids,” https://en.wikipedia.org/wiki/Boids, 2025, accessed: 2025-01-10.

APPENDIX

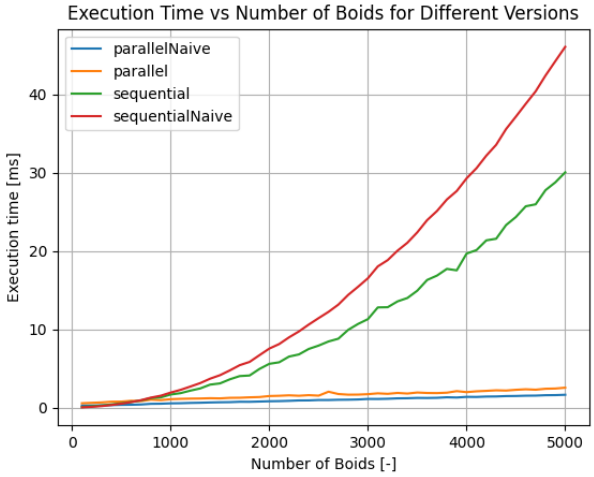
A. *Local Machine - Boid Count: 100–5 000*



(a) Plain



(b) Plain with Obstacles and Predators



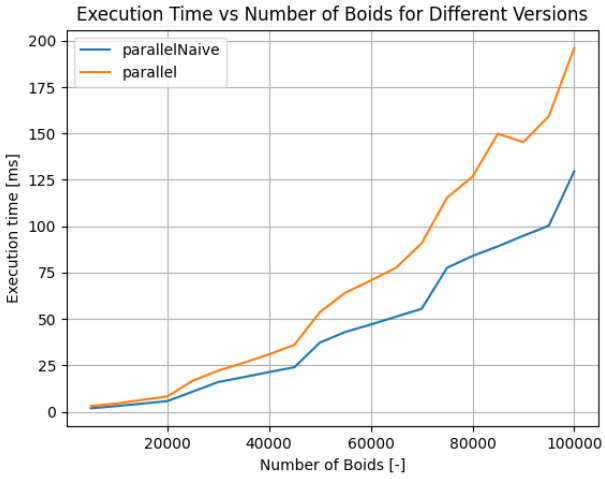
(c) 3D Complex

Figure 8: Execution time of a single simulation step for all implementation variants as a function of the number of spawned basic boids (100–5 000, step 100). Naive implementations are denoted by the suffix `naive`, while implementations without this suffix use grid-based neighborhood search. Measurements were performed on the *Local* machine, and each data point represents the average execution time over 15 simulation steps.

B. Local Machine - Boid Count: 5 000–1 000 000

(a) Plain

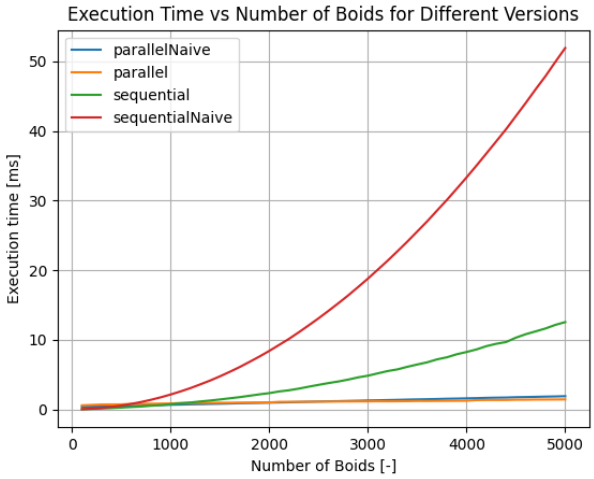
(b) Plain with Obstacles and Predators



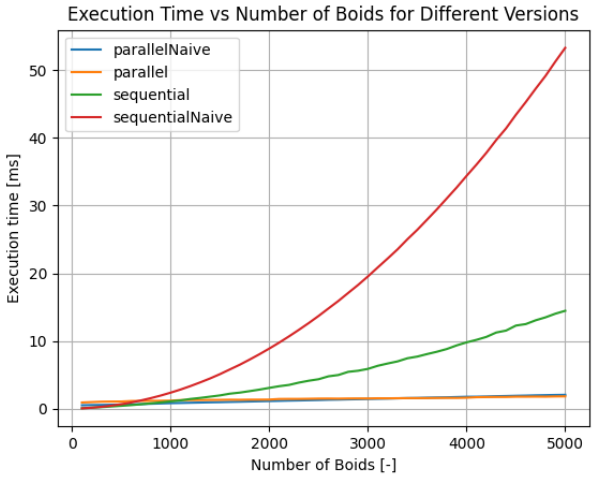
(c) 3D Complex

Figure 9: Execution time of a single simulation step for GPU implementation variants as a function of the number of spawned basic boids (5 000–100 000, step 5 000). The naive implementation is denoted by the suffix `naive`, while the implementation without this suffix uses grid-based neighborhood search. Measurements were performed on the *Local* machine, and each data point represents the average execution time over 15 simulation steps.

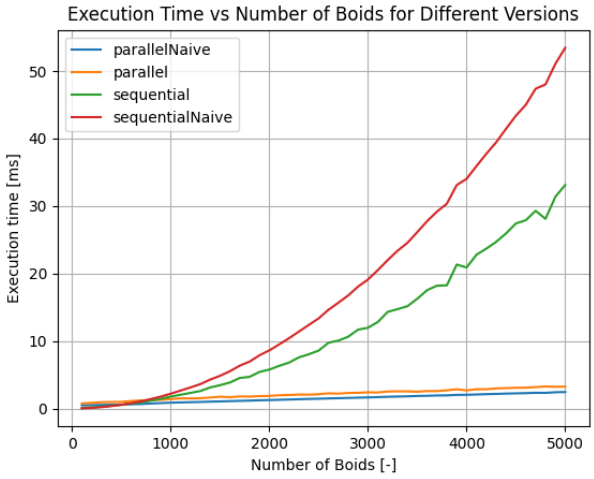
C. Server Machine - Boid Count: 100–5 000



(a) Plain



(b) Plain with Obstacles and Predators

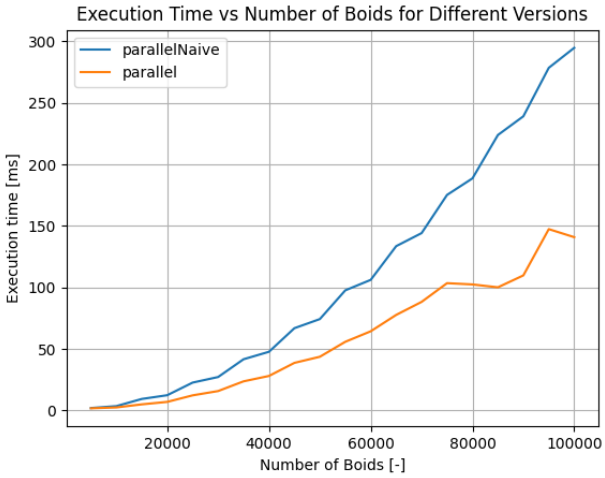


(c) 3D Complex

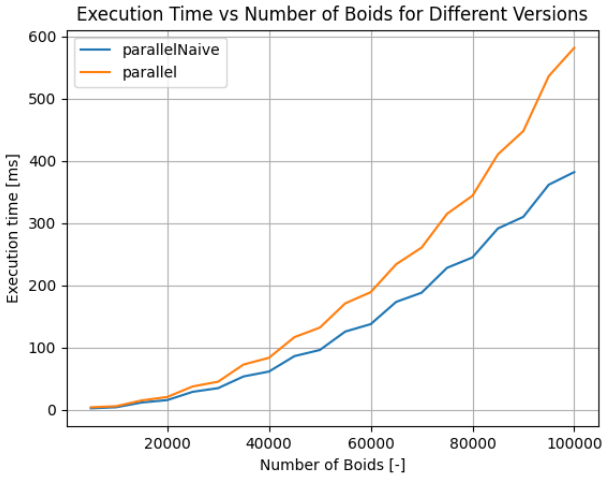
Figure 10: Execution time of a single simulation step for all implementation variants as a function of the number of spawned basic boids (100–5 000, step 100). Naive implementations are denoted by the suffix `naive`, while implementations without this suffix use grid-based neighborhood search. Measurements were performed on the *Server* machine, and each data point represents the average execution time over 15 simulation steps.

D. Server Machine - Boid Count: 5 000–1 000 000

(a) Plain



(b) Plain with Obstacles and Predators



(c) 3D Complex

Figure 11: Execution time of a single simulation step for GPU implementation variants as a function of the number of spawned basic boids (5 000–100 000, step 5 000). The naive implementation is denoted by the suffix `naive`, while the implementation without this suffix uses grid-based neighborhood search. Measurements were performed on the *Server* machine, and each data point represents the average execution time over 15 simulation steps.