Молдавский Государственный Университет

Факультет Математики и Информатики

Департамент Информатики

Лабораторная работа 3

Моделирование движения пешеходов

Выполнена студентом III курса

Группа I-2302-ru(ș.e.)

**Zabudico Alexandr**

Руководитель, преподаватель

**V.Trebiș**

Кишинёв, 2025

**Цель лабораторной работы:**

Исследовать поведение толпы людей при движении по ограниченному пространству.

**Алгоритм:**

* Используется алгоритм «Boids для пешеходов» с добавлением цели движения (Goal Seeking):
* Правила Boids: центрирование (держаться рядом с соседями),
* выравнивание (сĸорость и направление), разделение (избегание столĸновений).
* Новое правило: Движение ĸ цели (например, выход из здания).

**Подробные задачи:**

1. Создать 2D-среду с ĸоридорами, выходами и ограниченными зонами.

2. Создать набор агентов (пешеходов) с начальными позициями и сĸоростью.

3. Для ĸаждого агента на ĸаждом шаге:

• Найти ближайших соседей и рассчитать правила центрирования, выравнивания и разделения.

• Рассчитать веĸтор движения ĸ цели (выход).

• Суммировать все веĸторы с заданными весами.

• Ограничить сĸорость.

4. Визуализировать движение пешеходов, выделяя узĸие места и сĸопления.

5. Провести эĸсперименты:

• Изменять плотность людей.

• Менять вес правил (например, сильнее избегание столĸновений при высоĸой плотности).

• Анализировать, ĸаĸ группа реагирует на ограничения пространства.

**Код**

// js/Agent.js

/\*\*

 \* @class Agent

 \* @description Represents a pedestrian agent in the simulation.

 \* Each agent has position, velocity, and applies Boids rules plus goal seeking.

 \*/

class Agent {

  /\*\*

   \* @constructor

   \* @param {number} x - Initial x position.

   \* @param {number} y - Initial y position.

   \* @param {number} maxSpeed - Maximum speed (pixels per frame).

   \* @param {number} perceptionRadius - Radius to perceive neighbors.

   \* @param {number} desiredSeparation - Minimum distance to neighbors.

   \*/

  constructor(

    x,

    y,

    maxSpeed = 2,

    perceptionRadius = 50,

    desiredSeparation = 20

  ) {

*this*.position = new Vector(x, y);

*this*.velocity = new Vector(0, 0); // Initial velocity is zero

*this*.maxSpeed = maxSpeed;

*this*.perceptionRadius = perceptionRadius;

*this*.desiredSeparation = desiredSeparation;

*this*.radius = 5; // Visual radius for drawing

*this*.wallPerceptionRadius = 30; // Radius to perceive walls for avoidance

  }

  /\*\*

   \* Applies a force vector to the velocity.

   \* @param {Vector} force - The force to apply.

   \*/

  applyForce(force) {

*this*.velocity.add(force);

  }

  /\*\*

   \* Updates position based on velocity, limits speed, and handles boundaries/walls.

   \* @param {Environment} env - The environment for collision checks.

   \*/

  update(env) {

*this*.velocity.limit(*this*.maxSpeed);

    let nextPosition = Vector.add(*this*.position, *this*.velocity);

    // Soft correction if next position is invalid (fallback)

    if (!env.isPositionValid(nextPosition)) {

      // Instead of hard bounce, adjust velocity slightly away from wall

*this*.velocity.mult(0.8); // Reduce speed a bit

      nextPosition = Vector.add(*this*.position, *this*.velocity); // Recalculate

    }

*this*.position = nextPosition;

  }

  /\*\*

   \* Calculates separation force to avoid close neighbors.

   \* @param {Agent[]} neighbors - List of nearby agents.

   \* @returns {Vector} Separation steering vector.

   \*/

  separation(neighbors) {

    let steer = new Vector(0, 0);

    let count = 0;

    for (let neighbor of neighbors) {

      const d = *this*.position.dist(neighbor.position);

      if (d > 0 && d < *this*.desiredSeparation) {

        let diff = Vector.sub(*this*.position, neighbor.position);

        diff.normalize();

        diff.div(d); // Weight by inverse distance

        steer.add(diff);

        count++;

      }

    }

    if (count > 0) {

      steer.div(count);

    }

    if (steer.mag() > 0) {

      steer.normalize().mult(*this*.maxSpeed).sub(*this*.velocity);

    }

    return steer;

  }

  /\*\*

   \* Calculates alignment force to match neighbors' velocity.

   \* @param {Agent[]} neighbors - List of nearby agents.

   \* @returns {Vector} Alignment steering vector.

   \*/

  alignment(neighbors) {

    let sum = new Vector(0, 0);

    let count = 0;

    for (let neighbor of neighbors) {

      sum.add(neighbor.velocity);

      count++;

    }

    if (count > 0) {

      sum.div(count).normalize().mult(*this*.maxSpeed).sub(*this*.velocity);

      return sum;

    }

    return new Vector(0, 0);

  }

  /\*\*

   \* Calculates cohesion force towards neighbors' center.

   \* @param {Agent[]} neighbors - List of nearby agents.

   \* @returns {Vector} Cohesion steering vector.

   \*/

  cohesion(neighbors) {

    let sum = new Vector(0, 0);

    let count = 0;

    for (let neighbor of neighbors) {

      sum.add(neighbor.position);

      count++;

    }

    if (count > 0) {

      sum.div(count);

      let desired = Vector.sub(sum, *this*.position)

        .normalize()

        .mult(*this*.maxSpeed)

        .sub(*this*.velocity);

      return desired;

    }

    return new Vector(0, 0);

  }

  /\*\*

   \* Calculates goal-seeking force towards the exit.

   \* @param {Vector} goal - The goal position.

   \* @returns {Vector} Goal steering vector.

   \*/

  seekGoal(goal) {

    let desired = Vector.sub(goal, *this*.position)

      .normalize()

      .mult(*this*.maxSpeed);

    return desired.sub(*this*.velocity);

  }

  /\*\*

   \* Calculates avoidance force from nearby walls.

   \* @param {Environment} env - The environment with walls.

   \* @returns {Vector} Wall avoidance steering vector.

   \*/

  wallAvoidance(env) {

    let steer = new Vector(0, 0);

    let count = 0;

    // Check each wall

    for (let wall of env.walls) {

      // Find closest point on wall to agent

      let closestX = Math.max(

        wall.x,

        Math.min(*this*.position.x, wall.x + wall.w)

      );

      let closestY = Math.max(

        wall.y,

        Math.min(*this*.position.y, wall.y + wall.h)

      );

      let closestPoint = new Vector(closestX, closestY);

      const d = *this*.position.dist(closestPoint);

      if (d > 0 && d < *this*.wallPerceptionRadius) {

        let diff = Vector.sub(*this*.position, closestPoint);

        diff.normalize();

        diff.div(d); // Weight by inverse distance, stronger closer

        steer.add(diff);

        count++;

      }

    }

    // Also check restricted zones similarly

    for (let zone of env.restrictedZones) {

      let closestX = Math.max(

        zone.x,

        Math.min(*this*.position.x, zone.x + zone.w)

      );

      let closestY = Math.max(

        zone.y,

        Math.min(*this*.position.y, zone.y + zone.h)

      );

      let closestPoint = new Vector(closestX, closestY);

      const d = *this*.position.dist(closestPoint);

      if (d > 0 && d < *this*.wallPerceptionRadius) {

        let diff = Vector.sub(*this*.position, closestPoint);

        diff.normalize();

        diff.div(d);

        steer.add(diff);

        count++;

      }

    }

    if (count > 0) {

      steer.div(count);

    }

    if (steer.mag() > 0) {

      steer.normalize().mult(*this*.maxSpeed).sub(*this*.velocity);

    }

    return steer;

  }

}

// js/Environment.js

/\*\*

 \* @class Environment

 \* @description Represents the 2D environment with walls, exit, and boundaries.

 \* Uses simple rectangles for walls and exit.

 \*/

class Environment {

  /\*\*

   \* @constructor

   \* @param {number} width - Canvas width.

   \* @param {number} height - Canvas height.

   \*/

  constructor(width, height) {

*this*.width = width;

*this*.height = height;

*this*.walls = []; // Array of wall rectangles: {x, y, w, h}

*this*.exit = { x: width - 50, y: height / 2 - 25, w: 50, h: 50 }; // Green exit rectangle

*this*.restrictedZones = []; // Similar to walls, but could be no-entry areas

*this*.setupEnvironment();

  }

  /\*\*

   \* Sets up the environment with corridors and walls.

   \*/

  setupEnvironment() {

    // Example: Horizontal corridor with walls on top and bottom

    // Top wall

*this*.walls.push({ x: 0, y: 0, w: *this*.width, h: 20 });

    // Bottom wall

*this*.walls.push({ x: 0, y: *this*.height - 20, w: *this*.width, h: 20 });

    // Left wall

*this*.walls.push({ x: 0, y: 0, w: 20, h: *this*.height });

    // Narrow corridor in the middle

*this*.walls.push({ x: 300, y: 100, w: 20, h: 200 }); // Vertical wall creating bottleneck

*this*.walls.push({ x: 300, y: 400, w: 20, h: 200 }); // Another for corridor

    // Restricted zone example (e.g., no-entry area)

*this*.restrictedZones.push({ x: 100, y: 100, w: 100, h: 100 });

  }

  /\*\*

   \* Checks if a position is valid (not inside walls or restricted zones).

   \* @param {Vector} pos - Position to check.

   \* @returns {boolean} True if valid.

   \*/

  isPositionValid(pos) {

    const agentRadius = 5; // Buffer

    const agentRect = {

      x: pos.x - agentRadius,

      y: pos.y - agentRadius,

      w: agentRadius \* 2,

      h: agentRadius \* 2,

    };

    // Check walls

    for (let wall of *this*.walls) {

      if (*this*.rectIntersect(agentRect, wall)) {

        return false;

      }

    }

    // Check restricted zones

    for (let zone of *this*.restrictedZones) {

      if (*this*.rectIntersect(agentRect, zone)) {

        return false;

      }

    }

    // Check boundaries

    if (pos.x < 0 || pos.x > *this*.width || pos.y < 0 || pos.y > *this*.height) {

      return false;

    }

    return true;

  }

  /\*\*

   \* Checks if two rectangles intersect.

   \* @param {Object} r1 - First rect {x,y,w,h}.

   \* @param {Object} r2 - Second rect {x,y,w,h}.

   \* @returns {boolean} True if intersect.

   \*/

  rectIntersect(r1, r2) {

    return !(

      r1.x + r1.w < r2.x ||

      r1.x > r2.x + r2.w ||

      r1.y + r1.h < r2.y ||

      r1.y > r2.y + r2.h

    );

  }

  /\*\*

   \* Draws the environment on canvas.

   \* @param {CanvasRenderingContext2D} ctx - Canvas context.

   \*/

  draw(ctx) {

    // Draw walls (gray rectangles)

    ctx.fillStyle = "gray";

    for (let wall of *this*.walls) {

      ctx.fillRect(wall.x, wall.y, wall.w, wall.h);

    }

    // Draw restricted zones (red hatched or semi-transparent)

    ctx.fillStyle = "rgba(255, 0, 0, 0.3)";

    for (let zone of *this*.restrictedZones) {

      ctx.fillRect(zone.x, zone.y, zone.w, zone.h);

    }

    // Draw exit (green rectangle)

    ctx.fillStyle = "green";

    ctx.fillRect(*this*.exit.x, *this*.exit.y, *this*.exit.w, *this*.exit.h);

  }

  /\*\*

   \* Checks if position is at exit.

   \* @param {Vector} pos - Agent position.

   \* @param {number} threshold - Distance threshold.

   \* @returns {boolean} True if at exit.

   \*/

  isAtExit(pos, threshold = 10) {

    const exitCenter = new Vector(

*this*.exit.x + *this*.exit.w / 2,

*this*.exit.y + *this*.exit.h / 2

    );

    return pos.dist(exitCenter) < threshold;

  }

}

// js/main.js

/\*\*

 \* @description Main entry point for the simulation.

 \* Sets up event listeners, animation loop, and controls.

 \*/

// Get elements

const canvas = document.getElementById("canvas");

const statsEl = document.getElementById("stats");

const sim = new Simulation(canvas);

// Control elements

const numAgentsInput = document.getElementById("numAgents");

const numAgentsValue = document.getElementById("numAgentsValue");

const separationWeightInput = document.getElementById("separationWeight");

const separationWeightValue = document.getElementById("separationWeightValue");

const alignmentWeightInput = document.getElementById("alignmentWeight");

const alignmentWeightValue = document.getElementById("alignmentWeightValue");

const cohesionWeightInput = document.getElementById("cohesionWeight");

const cohesionWeightValue = document.getElementById("cohesionWeightValue");

const goalWeightInput = document.getElementById("goalWeight");

const goalWeightValue = document.getElementById("goalWeightValue");

const perceptionRadiusInput = document.getElementById("perceptionRadius");

const perceptionRadiusValue = document.getElementById("perceptionRadiusValue");

const resetButton = document.getElementById("resetButton");

// Update value displays

numAgentsInput.addEventListener(

  "input",

  () => (numAgentsValue.textContent = numAgentsInput.value)

);

separationWeightInput.addEventListener(

  "input",

  () => (separationWeightValue.textContent = separationWeightInput.value)

);

alignmentWeightInput.addEventListener(

  "input",

  () => (alignmentWeightValue.textContent = alignmentWeightInput.value)

);

cohesionWeightInput.addEventListener(

  "input",

  () => (cohesionWeightValue.textContent = cohesionWeightInput.value)

);

goalWeightInput.addEventListener(

  "input",

  () => (goalWeightValue.textContent = goalWeightInput.value)

);

perceptionRadiusInput.addEventListener(

  "input",

  () => (perceptionRadiusValue.textContent = perceptionRadiusInput.value)

);

// Reset button

resetButton.addEventListener("click", () => {

  const newParams = {

    numAgents: parseInt(numAgentsInput.value),

    separationWeight: parseFloat(separationWeightInput.value),

    alignmentWeight: parseFloat(alignmentWeightInput.value),

    cohesionWeight: parseFloat(cohesionWeightInput.value),

    goalWeight: parseFloat(goalWeightInput.value),

    perceptionRadius: parseInt(perceptionRadiusInput.value),

  };

  sim.reset(newParams);

});

// Animation loop

function animate() {

  sim.update();

  sim.draw();

  sim.updateStats(statsEl);

  requestAnimationFrame(animate);

}

animate();

// js/Simulation.js

/\*\*

 \* @class Simulation

 \* @description Manages the agents, environment, and simulation logic.

 \* Handles updates, neighbor searches, visualizations, and stats.

 \*/

class Simulation {

  /\*\*

   \* @constructor

   \* @param {HTMLCanvasElement} canvas - The canvas element.

   \*/

  constructor(canvas) {

*this*.canvas = canvas;

*this*.ctx = canvas.getContext("2d");

*this*.env = new Environment(canvas.width, canvas.height);

*this*.agents = [];

*this*.frame = 0;

*this*.maxDensity = 0;

*this*.jamCount = 0;

*this*.startTime = Date.now(); // For evacuation time

*this*.params = {

      numAgents: 50,

      separationWeight: 1.5,

      alignmentWeight: 1.0,

      cohesionWeight: 1.0,

      goalWeight: 1.2,

      perceptionRadius: 50,

    };

*this*.grid = null; // For spatial partitioning

*this*.gridSize = 50; // Bucket size for optimization

*this*.initAgents();

*this*.setupGrid();

  }

  /\*\*

   \* Initializes agents in a starting area.

   \*/

  initAgents() {

*this*.agents = [];

    const startArea = { x: 50, y: 50, w: 200, h: 500 }; // Starting room

    for (let i = 0; i < *this*.params.numAgents; i++) {

      let x, y;

      do {

        x = startArea.x + Math.random() \* startArea.w;

        y = startArea.y + Math.random() \* startArea.h;

      } while (!*this*.env.isPositionValid(new Vector(x, y)));

      const agent = new Agent(x, y, 2, *this*.params.perceptionRadius, 20);

*this*.agents.push(agent);

    }

  }

  /\*\*

   \* Sets up spatial grid for efficient neighbor search.

   \*/

  setupGrid() {

*this*.grid = [];

    const cols = Math.ceil(*this*.canvas.width / *this*.gridSize);

    const rows = Math.ceil(*this*.canvas.height / *this*.gridSize);

    for (let i = 0; i < cols; i++) {

*this*.grid[i] = [];

      for (let j = 0; j < rows; j++) {

*this*.grid[i][j] = [];

      }

    }

  }

  /\*\*

   \* Updates the spatial grid with current agents.

   \*/

  updateGrid() {

    // Clear grid

    for (let i = 0; i < *this*.grid.length; i++) {

      for (let j = 0; j < *this*.grid[i].length; j++) {

*this*.grid[i][j].length = 0;

      }

    }

    // Place agents in buckets

    for (let agent of *this*.agents) {

      const col = Math.floor(agent.position.x / *this*.gridSize);

      const row = Math.floor(agent.position.y / *this*.gridSize);

      if (

        col >= 0 &&

        col < *this*.grid.length &&

        row >= 0 &&

        row < *this*.grid[0].length

      ) {

*this*.grid[col][row].push(agent);

      }

    }

  }

  /\*\*

   \* Finds neighbors for an agent using spatial grid.

   \* @param {Agent} agent - The agent.

   \* @returns {Agent[]} List of neighbors within perception radius (limited to 10).

   \*/

  getNeighbors(agent) {

    const neighbors = [];

    const col = Math.floor(agent.position.x / *this*.gridSize);

    const row = Math.floor(agent.position.y / *this*.gridSize);

    const range = Math.ceil(agent.perceptionRadius / *this*.gridSize);

    for (let i = -range; i <= range; i++) {

      for (let j = -range; j <= range; j++) {

        const c = col + i;

        const r = row + j;

        if (

          c >= 0 &&

          c < *this*.grid.length &&

          r >= 0 &&

          r < *this*.grid[0].length

        ) {

          for (let other of *this*.grid[c][r]) {

            if (

              other !== agent &&

              agent.position.dist(other.position) < agent.perceptionRadius

            ) {

              neighbors.push(other);

              if (neighbors.length >= 10) return neighbors; // Limit to top 10

            }

          }

        }

      }

    }

    return neighbors;

  }

  /\*\*

   \* Updates all agents' forces and positions.

   \*/

  update() {

*this*.frame++;

*this*.updateGrid();

    let totalSpeed = 0;

    let jamThisFrame = 0;

    for (let i = *this*.agents.length - 1; i >= 0; i--) {

      const agent = *this*.agents[i];

      const neighbors = *this*.getNeighbors(agent);

      // Calculate forces

      const separation = agent

        .separation(neighbors)

        .mult(*this*.params.separationWeight);

      const alignment = agent

        .alignment(neighbors)

        .mult(*this*.params.alignmentWeight);

      const cohesion = agent

        .cohesion(neighbors)

        .mult(*this*.params.cohesionWeight);

      const goal = agent

        .seekGoal(

          new Vector(

*this*.env.exit.x + *this*.env.exit.w / 2,

*this*.env.exit.y + *this*.env.exit.h / 2

          )

        )

        .mult(*this*.params.goalWeight);

      // Wall avoidance force

      const wallAvoid = agent.wallAvoidance(*this*.env).mult(1.5); // Weight for wall avoidance

      // Apply combined force

      const force = new Vector(0, 0)

        .add(separation)

        .add(alignment)

        .add(cohesion)

        .add(goal)

        .add(wallAvoid);

      agent.applyForce(force);

      // Update position

      agent.update(*this*.env);

      // Check for exit

      if (*this*.env.isAtExit(agent.position)) {

*this*.agents.splice(i, 1);

      }

      totalSpeed += agent.velocity.mag();

      // Check for jams (speed < 0.5)

      if (agent.velocity.mag() < 0.5) {

        jamThisFrame++;

      }

    }

    // Update stats

    const density = *this*.calculateMaxDensity();

*this*.maxDensity = Math.max(*this*.maxDensity, density);

*this*.jamCount += jamThisFrame > 0 ? 1 : 0; // Count frames with jams

    // Check if all evacuated

    if (*this*.agents.length === 0) {

      const evacTime = (Date.now() - *this*.startTime) / 1000; // Seconds

      console.log(

        `Evacuation time: ${*this*.frame} frames (${evacTime} seconds)`

      );

      console.log(`Max density observed: ${*this*.maxDensity}`);

      console.log(`Jam count (frames with low speed): ${*this*.jamCount}`);

    }

    // Average speed for stats

*this*.averageSpeed =

*this*.agents.length > 0 ? totalSpeed / *this*.agents.length : 0;

  }

  /\*\*

   \* Calculates current max density (agents per unit area in clusters).

   \* @returns {number} Max density.

   \*/

  calculateMaxDensity() {

    let maxD = 0;

    for (let agent of *this*.agents) {

      const neighbors = *this*.getNeighbors(agent);

      const localD = neighbors.length / (Math.PI \* agent.perceptionRadius \*\* 2); // Density

      maxD = Math.max(maxD, localD);

    }

    return maxD;

  }

  /\*\*

   \* Draws everything on the canvas.

   \*/

  draw() {

*this*.ctx.clearRect(0, 0, *this*.canvas.width, *this*.canvas.height);

*this*.env.draw(*this*.ctx);

    // Draw agents (blue circles)

*this*.ctx.fillStyle = "blue";

    for (let agent of *this*.agents) {

*this*.ctx.beginPath();

*this*.ctx.arc(

        agent.position.x,

        agent.position.y,

        agent.radius,

        0,

        2 \* Math.PI

      );

*this*.ctx.fill();

    }

    // Highlight bottlenecks (red circles where >5 agents in 30px)

    // And crowds (yellow overlays for dense clusters)

*this*.highlightClusters();

  }

  /\*\*

   \* Highlights dense areas and bottlenecks.

   \*/

  highlightClusters() {

    for (let agent of *this*.agents) {

      const neighbors = *this*.getNeighbors(agent);

      if (neighbors.length > 5) {

        const clusterRadius = 30;

        // Yellow overlay for crowd

*this*.ctx.fillStyle = "rgba(255, 255, 0, 0.3)";

*this*.ctx.beginPath();

*this*.ctx.arc(

          agent.position.x,

          agent.position.y,

          clusterRadius,

          0,

          2 \* Math.PI

        );

*this*.ctx.fill();

        // Red circle for bottleneck if in narrow area (simple: near walls)

        if (*this*.isNearWall(agent.position)) {

*this*.ctx.strokeStyle = "red";

*this*.ctx.lineWidth = 2;

*this*.ctx.beginPath();

*this*.ctx.arc(

            agent.position.x,

            agent.position.y,

            clusterRadius,

            0,

            2 \* Math.PI

          );

*this*.ctx.stroke();

        }

      }

    }

  }

  /\*\*

   \* Checks if position is near a wall (within 20px).

   \* @param {Vector} pos - Position.

   \* @returns {boolean} True if near wall.

   \*/

  isNearWall(pos) {

    for (let wall of *this*.env.walls) {

      if (

        Math.min(

          Math.abs(pos.x - wall.x),

          Math.abs(pos.x - (wall.x + wall.w))

        ) < 20 ||

        Math.min(

          Math.abs(pos.y - wall.y),

          Math.abs(pos.y - (wall.y + wall.h))

        ) < 20

      ) {

        return true;

      }

    }

    return false;

  }

  /\*\*

   \* Resets the simulation with new parameters.

   \* @param {Object} newParams - New parameter values.

   \*/

  reset(newParams) {

    Object.assign(*this*.params, newParams);

*this*.frame = 0;

*this*.maxDensity = 0;

*this*.jamCount = 0;

*this*.startTime = Date.now();

*this*.initAgents();

  }

  /\*\*

   \* Updates the stats display.

   \* @param {HTMLElement} statsEl - The stats DOM element.

   \*/

  updateStats(statsEl) {

    statsEl.innerHTML = `

            Frame: ${*this*.frame}<br>

            Agents Left: ${*this*.agents.length}<br>

            Average Speed: ${*this*.averageSpeed.toFixed(2)}

        `;

  }

}

// js/Vector.js

/\*\*

 \* @class Vector

 \* @description Represents a 2D vector with operations for Boids calculations.

 \* Provides methods for addition, subtraction, scaling, normalization, etc.

 \*/

class Vector {

  /\*\*

   \* @constructor

   \* @param {number} x - The x-component.

   \* @param {number} y - The y-component.

   \*/

  constructor(x, y) {

*this*.x = x;

*this*.y = y;

  }

  /\*\*

   \* Adds another vector to this one.

   \* @param {Vector} v - The vector to add.

   \* @returns {Vector} This vector after addition.

   \*/

  add(v) {

*this*.x += v.x;

*this*.y += v.y;

    return *this*;

  }

  /\*\*

   \* Subtracts another vector from this one.

   \* @param {Vector} v - The vector to subtract.

   \* @returns {Vector} This vector after subtraction.

   \*/

  sub(v) {

*this*.x -= v.x;

*this*.y -= v.y;

    return *this*;

  }

  /\*\*

   \* Scales this vector by a scalar.

   \* @param {number} s - The scalar value.

   \* @returns {Vector} This vector after scaling.

   \*/

  mult(s) {

*this*.x \*= s;

*this*.y \*= s;

    return *this*;

  }

  /\*\*

   \* Divides this vector by a scalar.

   \* @param {number} s - The scalar value (must not be zero).

   \* @returns {Vector} This vector after division.

   \*/

  div(s) {

    if (s !== 0) {

*this*.x /= s;

*this*.y /= s;

    }

    return *this*;

  }

  /\*\*

   \* Calculates the magnitude (length) of this vector.

   \* @returns {number} The magnitude.

   \*/

  mag() {

    return Math.sqrt(*this*.x \* *this*.x + *this*.y \* *this*.y);

  }

  /\*\*

   \* Normalizes this vector to unit length.

   \* @returns {Vector} This vector after normalization.

   \*/

  normalize() {

    const m = *this*.mag();

    if (m !== 0) {

*this*.div(m);

    }

    return *this*;

  }

  /\*\*

   \* Limits the magnitude of this vector to a maximum value.

   \* @param {number} max - The maximum magnitude.

   \* @returns {Vector} This vector after limiting.

   \*/

  limit(max) {

    const m = *this*.mag();

    if (m > max) {

*this*.normalize().mult(max);

    }

    return *this*;

  }

  /\*\*

   \* Creates a copy of this vector.

   \* @returns {Vector} A new Vector instance with the same components.

   \*/

  copy() {

    return new Vector(*this*.x, *this*.y);

  }

  /\*\*

   \* Calculates the distance to another vector.

   \* @param {Vector} v - The other vector.

   \* @returns {number} The Euclidean distance.

   \*/

  dist(v) {

    const dx = *this*.x - v.x;

    const dy = *this*.y - v.y;

    return Math.sqrt(dx \* dx + dy \* dy);

  }

  /\*\*

   \* Static method to add two vectors.

   \* @param {Vector} a - First vector.

   \* @param {Vector} b - Second vector.

   \* @returns {Vector} A new vector sum.

   \*/

  static add(a, b) {

    return new Vector(a.x + b.x, a.y + b.y);

  }

  /\*\*

   \* Static method to subtract two vectors.

   \* @param {Vector} a - First vector.

   \* @param {Vector} b - Second vector.

   \* @returns {Vector} A new vector difference.

   \*/

  static sub(a, b) {

    return new Vector(a.x - b.x, a.y - b.y);

  }

}

<!-- index.html -->

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Pedestrian Movement Modeling Simulation</title>

    <link href="https://fonts.googleapis.com/css2?family=Roboto:wght@300;400;700&display=swap" rel="stylesheet">

    <style>

        body {

            margin: 0;

            font-family: 'Roboto', sans-serif;

            background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);

            color: #333;

            padding: 20px;

        }

        #canvas {

            border: 2px solid #4A90E2;

            border-radius: 8px;

            box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);

            display: block;

            margin: 0 auto;

            background: white;

        }

        #stats {

            position: absolute;

            top: 30px;

            left: 30px;

            background: rgba(255, 255, 255, 0.9);

            padding: 15px;

            border-radius: 10px;

            box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);

            font-size: 14px;

            color: #333;

        }

        #controls {

            max-width: 800px;

            margin: 20px auto;

            padding: 20px;

            background: white;

            border-radius: 10px;

            box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);

        }

        #controls label {

            display: block;

            margin-bottom: 10px;

            font-weight: 700;

            color: #4A90E2;

        }

        #controls input[type="range"] {

            width: 100%;

            -webkit-appearance: none;

            height: 5px;

            border-radius: 5px;

            background: #ddd;

            outline: none;

            margin-bottom: 5px;

        }

        #controls input[type="range"]::-webkit-slider-thumb {

            -webkit-appearance: none;

            appearance: none;

            width: 20px;

            height: 20px;

            border-radius: 50%;

            background: #4A90E2;

            cursor: pointer;

            box-shadow: 0 2px 4px rgba(0, 0, 0, 0.2);

        }

        #controls input[type="range"]::-moz-range-thumb {

            width: 20px;

            height: 20px;

            border-radius: 50%;

            background: #4A90E2;

            cursor: pointer;

            box-shadow: 0 2px 4px rgba(0, 0, 0, 0.2);

        }

        #controls button {

            background: linear-gradient(45deg, #4A90E2, #357ABD);

            color: white;

            border: none;

            padding: 12px 24px;

            border-radius: 6px;

            cursor: pointer;

            font-size: 16px;

            transition: all 0.3s ease;

            box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);

            display: block;

            margin: 20px auto 0;

        }

        #controls button:hover {

            transform: translateY(-2px);

            box-shadow: 0 4px 8px rgba(0, 0, 0, 0.2);

        }

        .value-span {

            font-weight: 400;

            color: #666;

            margin-left: 10px;

        }

    </style>

</head>

<body>

    <canvas id="canvas" width="800" height="600"></canvas>

    <div id="stats"></div>

    <div id="controls">

        <label>Number of Agents: <span id="numAgentsValue" class="value-span">50</span></label>

        <input type="range" id="numAgents" min="10" max="200" value="50"><br>

        <label>Separation Weight: <span id="separationWeightValue" class="value-span">1.5</span></label>

        <input type="range" id="separationWeight" min="0.5" max="2.0" step="0.1" value="1.5"><br>

        <label>Alignment Weight: <span id="alignmentWeightValue" class="value-span">1.0</span></label>

        <input type="range" id="alignmentWeight" min="0.5" max="2.0" step="0.1" value="1.0"><br>

        <label>Cohesion Weight: <span id="cohesionWeightValue" class="value-span">1.0</span></label>

        <input type="range" id="cohesionWeight" min="0.5" max="2.0" step="0.1" value="1.0"><br>

        <label>Goal Weight: <span id="goalWeightValue" class="value-span">1.2</span></label>

        <input type="range" id="goalWeight" min="0.5" max="2.0" step="0.1" value="1.2"><br>

        <label>Perception Radius: <span id="perceptionRadiusValue" class="value-span">50</span></label>

        <input type="range" id="perceptionRadius" min="20" max="100" value="50"><br>

        <button id="resetButton">Reset Simulation</button>

    </div>

    <script src="js/Vector.js"></script>

    <script src="js/Agent.js"></script>

    <script src="js/Environment.js"></script>

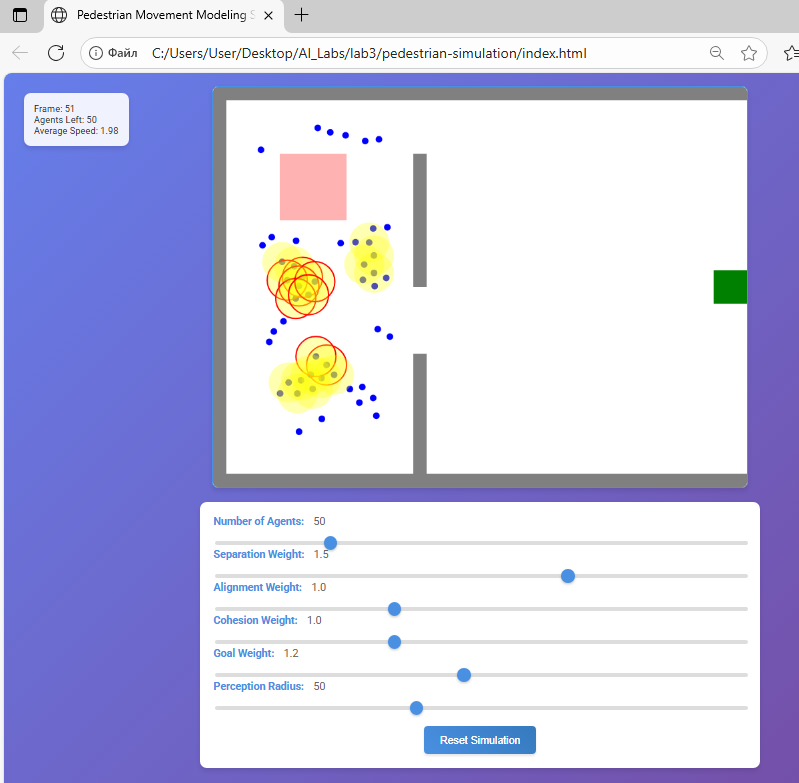
    <script src="js/Simulation.js"></script>

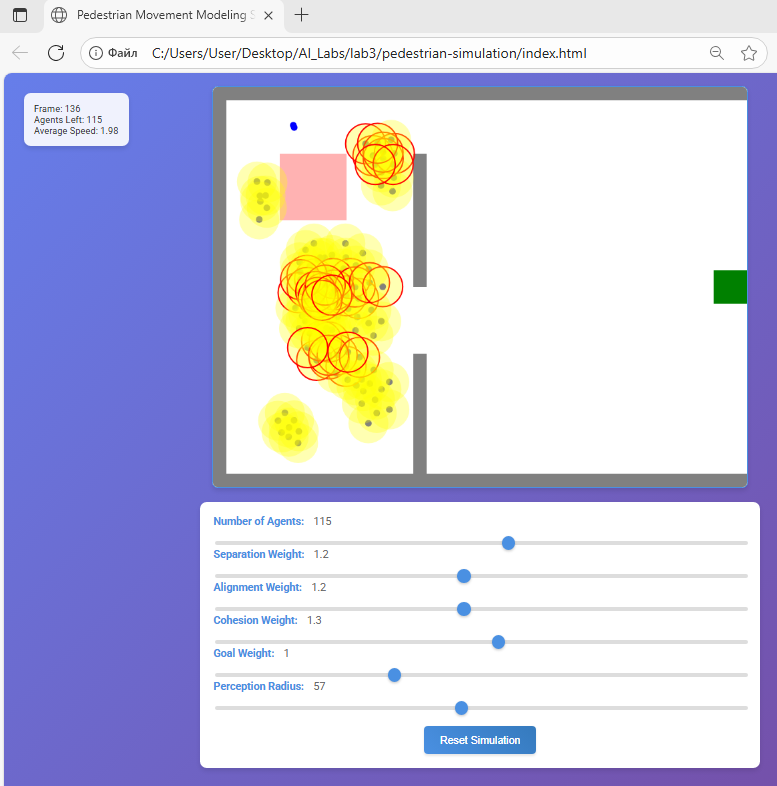
    <script src="js/main.js"></script>

</body>

</html>

**Результат**

****

****